COMP 348: ASSIGNMENT 1

Note that assignments must be submitted on time in order to receive full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate.

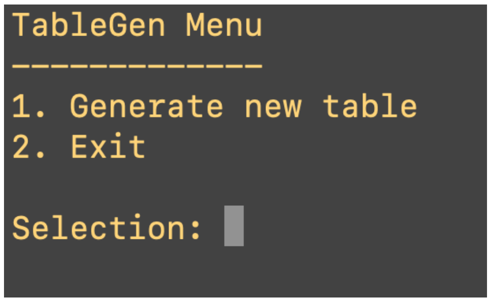
THE C PROGRAMMING LANGUAGE

DESCRIPTION: In this assignment, you will gain hands-on experience with the C programming language. While you are not required to be a C expert to complete the work, you will certainly have the opportunity to explore most of the things that we have discussed in class (and a few other things as well).

NOTE: This is a long document but the assignment itself is quite easy to describe. Most of the document consists of hints and advice that will make your life a little easier when writing your first C program.

In terms of your task, you will be creating a simple application that will generate data for a single output file. In practice, such applications are often used to create realistic test sets for databases or other data management systems. Robust, commercial data generators can be quite sophisticated, as they are often used to produce data for many tables, with properties and requirements that can be challenging to support. Our application, of course, will be MUCH simpler. It will work as follows.

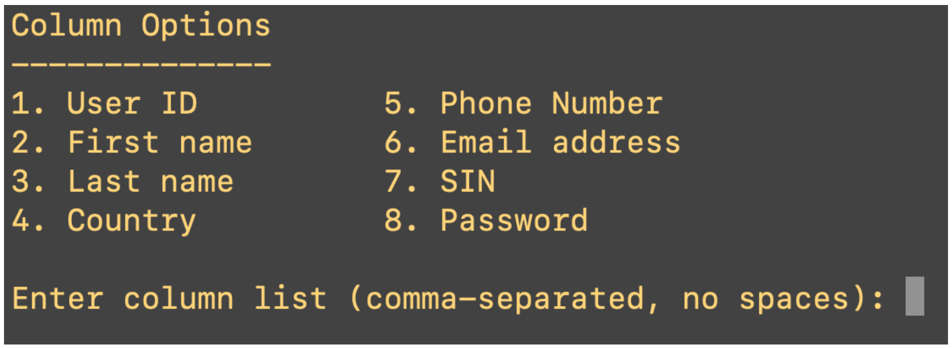
To begin, you will display a trivial starting menu that will contain just two options: build a table and exit. When run, the program will look like this:



```
TableGen Menu
-------------
1. Generate new table
2. Exit

Selection:
```

Selecting Option 2 will, of course, exit the program and display a simple message to say "Goodbye and thanks for using TableGen".

If Option 1 is selected, on the other hand, a new menu will be displayed. This new menu will look like this:



```
Column Options
--------------
1. User ID          5. Phone Number
2. First name       6. Email address
3. Last name        7. SIN
4. Country          8. Password

Enter column list (comma-separated, no spaces):
```

Before we go any further, note that the screen should always be cleared before either menu is displayed. In other word, the text should not simply keep scrolling down the screen. Doing this is quite easy – we simply use the `system` function found in `stdlib.h`. Specifically, `system("clear")` will clear the screen on a Linux system (e.g., the docker-based installation that the graders will be using.)

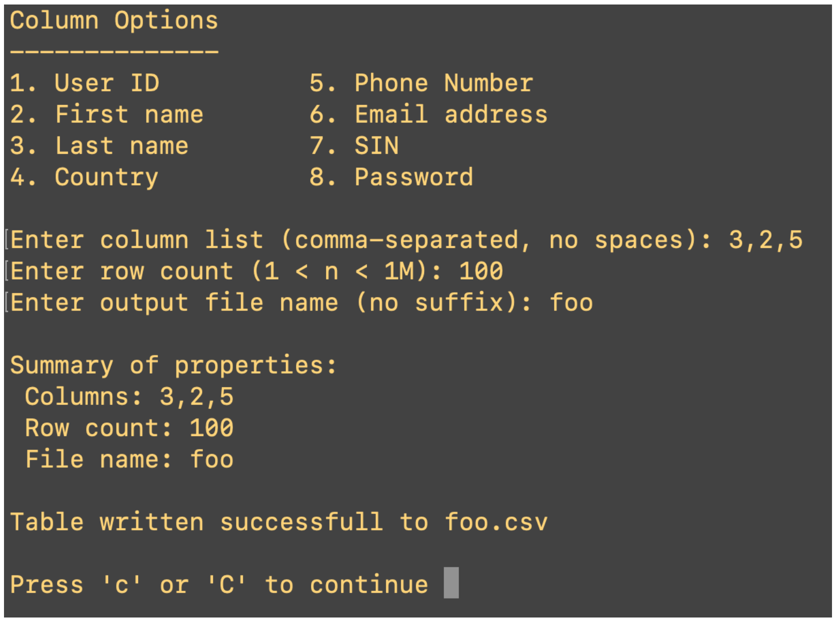
In terms of the new Column Options menu, it displays a list of columns that the user can select. The user is free to select any number of columns to be used for the table. Moreover, they may select them in any order (i.e., the order in which the columns will be displayed in the final table). Finally, note that the first column represents the "sort" column. In other words, the rows in the table will be sorted on this column before being written to a file.

The columns should be entered in a column-separated list, with no spaces. So, for example, `3,2,5` or `1,3,6,8,7`. It is important to use this format, as keyboard input in C can be somewhat unforgiving and parsing the column list can be quite tedious if, for example, spaces are used.

Once the columns are entered, a second prompt will ask the user to enter the row count. This is just an integer from 1 to 1 million. A third prompt will then ask the user for the name of the output file that will store the data.

That's it for input to the program. Your application will now generate the output and confirm that the table has been written. It will also provide a "continue" prompt (i.e., press c to continue). This will allow the user to go back to the main menu, where they can choose to either exit the program or generate a new table.

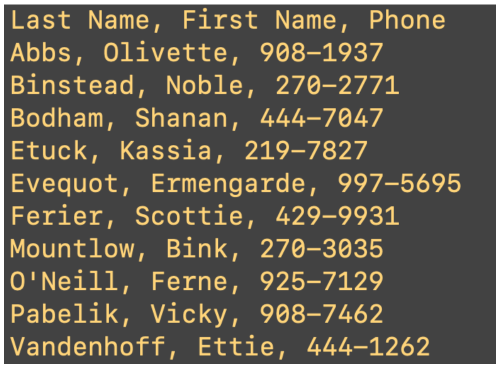
The image below shows the screen output once the input has been provided and the output has been written.



```
Column Options
--------------
1. User ID          5. Phone Number
2. First name       6. Email address
3. Last name        7. SIN
4. Country          8. Password

Enter column list (comma-separated, no spaces): 3,2,5
Enter row count (1 < n < 1M): 100
Enter output file name (no suffix): foo

Summary of properties:
 Columns: 3,2,5
 Row count: 100
 File name: foo

Table written successfull to foo.csv

Press 'c' or 'C' to continue
```

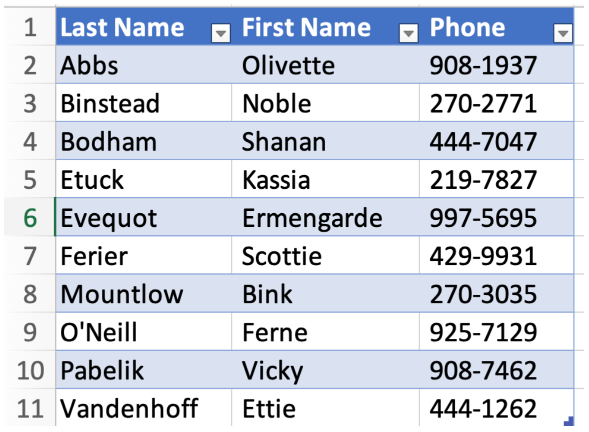
To summarize, the user has asked to create a table with 3 columns: last name, first name, and phone number. It will have 100 rows and will be written to a file called foo.csv.

What does foo.csv look like? It is simply a text file in which the column values are separated by commas. These comma-delimited csv text files are useful because they can be directly imported into spreadsheet applications line MS Excel for more convenient viewing (they can also be displayed directly to the screen as simple text). So let's look at the text version of foo.csv first.



```
Last Name, First Name, Phone
Abbs, Olivette, 908-1937
Binstead, Noble, 270-2771
Bodham, Shanan, 444-7047
Etuck, Kassia, 219-7827
Evequot, Ermengarde, 997-5695
Ferier, Scottie, 429-9931
Mountlow, Bink, 270-3035
O'Neill, Ferne, 925-7129
Pabelik, Vicky, 908-7462
Vandenhoff, Ettie, 444-1262
```

…and now the excel version (nicely formatted as a table):



	Last Name	First Name	Phone
1	Last Name	First Name	Phone
2	Abbs	Olivette	908-1937
3	Binstead	Noble	270-2771
4	Bodham	Shanan	444-7047
5	Etuck	Kassia	219-7827
6	Evequot	Ermengarde	997-5695
7	Ferier	Scottie	429-9931
8	Mountlow	Bink	270-3035
9	O'Neill	Ferne	925-7129
10	Pabelik	Vicky	908-7462
11	Vandenhoff	Ettie	444-1262

Let's examine the content, noting the following points:

1. The csv file contains a "header" row that provides the name of each column (e.g., "Last Name".
2. The columns are listed in the order defined by the user (3,2,5 = Last Name, First Name, and Phone).
3. The table is sorted (alphabetically) by last name.
4. Phone numbers are formatted in the standard way (xxx-xxxx).

So that's it in terms of the what the application does.

DATA GENERATION

The description of the application is quite simple – generate a table with rows of realistic looking data. But you will need a little more info before proceeding.

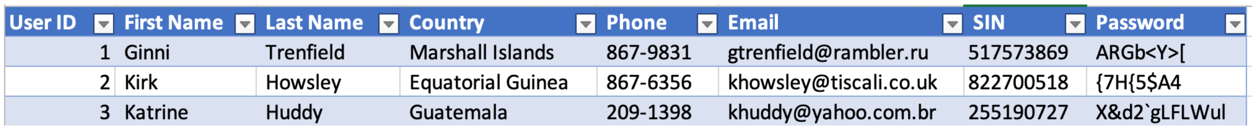
The most obvious question is "where does the data come from?" Below, we describe how the data will be generated for each column.

1. **User ID:** This is a simple integer, starting from 1. It is auto-incremented so that the first row contains the number 1, the second row will be 2, and so on. This one couldn't be simpler.
2. **First Name**: These values will come from a simple text file called `first_names.txt` that will be included with the assignment description. The file will have exactly 1000 entries and will consist of one entry per line. For example, the `first_names.txt` file will have entries like `Joe` and `Susan`. You will simply read the content of each file into a data structure (e.g., an array of strings) and then you will randomly pick one name to be used in the current row.
3. **Last Name**: Same mechanism as first name. The file will be called `last_names.txt` and will also have exactly 1000 entries.
4. **Country**: Same mechanism as first name. The file will be called `countries.txt` and will have exactly 195 entries.
5. **Phone Number**: This is a combination of the phone *index* (the first 3 digits) and phone *line* (the last 4 digits). The 10 valid index codes are [398, 270, 925, 867, 209, 429, 908, 997,

444, 219]. You can simply store these in an array in your program. For each row, you will randomly select one of these and then randomly generate the last 4 digits.

6. **Email address**: These are a little more complicated. They will consist of the first letter of the first name + the last name + @ + a randomly selected email suffix. The suffix will come from a text file called `email_suffixes.txt` and will have exactly 100 entries. So, for example, if we have someone named Joe Smith, the email address might be jsmith@hotmail.com. Note that capitalization does not matter. Also note that in order to create the email column, last name and first name must also be in the list of selected columns. To make things easier, you can assume that the graders will also ensure that this is the case if email is selected as one of the options. In other words, you don't have to verify this in the application.
7. **SIN** (social insurance number): This is a randomly generated 9-digit number.
8. **Password**: This is a text string, from 6 to 16 characters in length. It consists of all printable (non space) characters on a standard keyboard (i.e., letters, digits, and punctuation symbols). Just in case it's not obvious, you must exclude the comma (',') since you are generating a csv output file in which commas are used to separate the fields. Also note that, in C, chars are just numbers that represent numeric codes in the ASCII char set. So all you are really doing here is assigning relevant ascii codes to each char in the password.

Just for clarity, let's look at a few rows that include all 8 fields.



User ID	First Name	Last Name	Country	Phone	Email	SIN	Password
1	Ginni	Trenfield	Marshall Islands	867-9831	gtrenfield@rambler.ru	517573869	ARGb<Y>[
2	Kirk	Howsley	Equatorial Guinea	867-6356	khowsley@tiscali.co.uk	822700518	{7H{5$A4
3	Katrine	Huddy	Guatemala	209-1398	khuddy@yahoo.com.br	255190727	X&d2`gLFLWul

Here we see an example of all columns, each formatted in the required way. Note that we are using the first column, User ID, as the sort column, but since you are generating this as a integer sequence it should already be in sorted order.

THINGS TO KEEP IN MIND

If you were writing this application with a language like Python, this would be a very easy assignment, since there is no complex logic and Python has functions to do all of the generation, sorting, and storage in simple, easy to use methods.

C, on the other hand, will make things a little more difficult. Because we want you to focus on the structure of C, and the things that distinguish it from other languages, we do not want you to get lost in obscure issues and common "gotchas". So the following section gives you some advice about what to do and what to avoid. Please read this carefully!

- Start small. Do not try to do everything at once. You might, for example, get a simple menu to display first, then try to read one of the input files, then perhaps try to generate a single field, and so on.

- Compile frequently to see what's working and what is not. Do not continue until you have fixed the current problems (they will not magically go away in the future).
- If you have segmentation faults due to pointer problems (and you will), do NOT try to find them with print statements. This almost never works. Instead, use the `valgrind` system included with the docker image. An invocation like `valgrind --leak-check=yes my_app_name` will run and analyze your program. It will list problems along with the associated line numbers. To use `valgrind` effectively, you must compile your program with some additional options. On the docker installation, this would look like the following: `gcc -Wall -g -gdwarf-4 file1.c file2.c`
- We have kept the GUI as simple as possible. While it's possible to use several mechanisms to read keyboard input, the `scanf` function is probably the simplest since it reads the text and transforms it into the appropriate data type in one step. One non-obvious complication is that when reading the 'c' character to continue processing after the table has been generated, the `scanf` function will actually read the \n (newline) character that remains in the input buffer after the previous `scanf` call. As a result, you will probably want to use code like the following when capturing the c/C character:

```
char proceed;
printf("\nPress 'c' or 'C' to continue ");
do{
      scanf("%c", &proceed);
} while ( (proceed != 'c') && (proceed != 'C'));
```

  This will swallow any *newline* chars and then read the user's `c` character.

- Reading the input files is fairly straightforward. You will use `fopen` to open the input file(s) in read mode ("r"). You can then use the `fgets` function to read each line as a string (i.e., a `while` loop of `fgets` calls). None of the string names are very long so they can be read into a fixed size char array, something like `char buffer[64]`. The one non-obvious part is that `fgets` will included the \n character at the end of the string, which you definitely do not want (e.g., "joe\n", instead of "joe"). Since you need to transfer each string from the char buffer so that you can create an array of strings (that can be used for random selection), you need to make sure that you do not include the newline character in the final string. If you are using a function like `strdup` to create a copy of each string, you should think about using `strndup` instead, so that you can restrict the number of characters that are copied (i.e., not the trailing \n).
- Writing the output file(s) should also be fairly straightforward. Again, you will use `fopen` to open the file in "wt" (write text) mode. This will automatically over-write any existing file with that name (this is fine). You will probably want to use `fputs` to write strings directly to the csv file (either a string representing an entire row, or a sequence of strings for each column). If the fields are already text (e.g., last name), `fputs` can be used directly. If the field is an integer (e.g., user ID), then you can use a function like `sprintf` to convert it to a string and store it in a char array, and then use `fputs` to write the converted value from the char array to the file.

- You will likely be using pointers in this application. Note that a variable holding a pointer is typically declared like this: `int *foo`. Here, `foo` is a pointer to an int (or possibly an array of ints). So `foo[3]` could be used to access the third int in such an array.

  It is also possible to *dereference* a pointer, which means that we are getting the data that the pointer references. So syntax like `int x = *foo` implies that the int pointed to by `foo` will now be assigned to x. This is a technique that is typically used in the comparison function used when sorting arrays in C.
- In terms of sorting, you will used the `qsort` library function included in the standard libraries. The syntax is sometimes confusing for those not used to C. The basic idea is that `qsort` is a "generic" library function that sorts arrays of elements, based upon a comparison function provided by the programmer. As parameters, `qsort` takes (1) the array to be sorted, (2) the numbers of elements in the array, (3) the size (in bytes) of each element, and (4) the name of the comparison function.

  In turn, the comparison function takes two parameters: `void` pointers to any two array elements that `qsort` will compare during the sort. `qsort` uses "void" pointers since it doesn't know *a priori* what you want to sort. So, typically, the programmer `casts` the pointer to the right type (e.g., void* becomes int*) and then dereferences the pointer to get the element itself. The two elements can then be directly compared for equality. A quick look at a trivial `qsort` example will show these mechanisms in practice.

- To extract the columns from the user's list of columns (e.g., 5,4,3), you will use the `strtok` function in the standard string library. One simply uses a trivial while loop to separate the string on the comma delimiter. There are lots of simple examples of its use.
- While we generally avoid *global* variables that are visible throughout the whole program, you may find it convenient to use some globals in this application. That's OK. When multiple source files are being used, the compiler must be told that a shared global variable is actually defined somewhere else. For example, if `file1.c` contains a global variable defined as `int foo = 4;` then `file2.c` would include a declaration like `extern int foo`. This tells the compiler that the `foo` variable declared in `file2.c` is not a new variable with the same name (which would produce a compiler error) but is just a reference to the `foo` variable already defined and initialized in `file1.c`
- When multiple source files are used, header files should be employed in order to provide prototype information for the compiler. Never include variables or function implementations in a header file. NEVER! They are primarily used for function prototypes, constants, and things like struct definitions.
- Use the #define mechanism for any fixed values (i.e., constants) that should be used throughout the program.
- In terms of dynamically created data, the `malloc` functions are used for this purpose. They always return a pointer/address value.
- When de-allocating memory, you will the `free` function. To be clear, only use free with a matching `malloc` call. So a char array declared as `char *buffer = (char`

`*)malloc(10)` could later be deallocated with `free(buffer).` NEVER try to deallocate a char buffer (or any array) defined like this: `char buffer[10.` malloc was not used to create this and its memory is fully managed by the language environment.

- If a struct is defined like `struct foo {int x; inty;};` then we can declare a variable of this type as `struct foo bar`. We can now use `bar.x` or `bar.y` to access the internal values. But if we use `malloc` to dynamically create space for a new `struct foo` value, like `struct foo *bar = (struct foo *)malloc(sizeof(struct foo)),` then we must use the syntax `bar->x` or `bar->y` to access the value since we are now using a pointer to the foo struct.
- Using pointers is crucial in C but it can be confusing at first. A couple of things to keep in mind: If we use `malloc` to create an array (e.g., of ints or chars), we can just use the standard array syntax (e.g., `foo[4]`) to access the elements of the array. The C compiler already knows that the array variable is a pointer. It is also possible to have pointers to pointers. So if we use `malloc` to create an array of strings called `foo`, this would actually consist of a pointer to the main array, which would then contain pointers for the strings. So `foo` would be defined as `char **foo (or char* *foo)`. In other words, `foo` is pointer to an array of string pointers. This can be confusing at first and can lead to lots of unexpected segmentation faults if you are referencing the wrong pointer. Moreover, if you are deallocating this memory, you have to be REALLY careful. Specifically, the strings must be deallocated first, and then the main array. And, just to be clear, free'ing the main array does not automatically free the memory used by the strings.

EVALUATION: Please note that the markers will be using a standard Linux system, likely the same docker installation that most of you should be using. Your code MUST compile and run from the Linux command line. If you are using an IDE (e.g., Eclipse or VS Code), rather than a simple editor, you will want to test execution from the command line, since IDE's often create their own paths, folders, environment variables that might prevent the code from running on the command line.

To evaluate the submissions, the marker will simply create a test folder and add the input data files and your source files. Your code will be compiled and run and a few simple csv files will be created. Every student will be graded the same way. You are expected to deallocate any memory that you have allocated during the running of the application.

The markers will be given a simple spreadsheet that lists the various criteria described above. There are no mystery requirements. Please note that it is better to have a working version of a slightly restricted program than a non-working version of something that tried to do everything (e.g., you can generate some columns but not all of them, or you can write a non-sorted output file but not a sorted file). So make sure that your code actually compiles and runs. It is virtually impossible to grade an assignment that does not run at all.

DELIVERABLES: Your submission should have multiple source files (if you can't get that to work, one large source file will still be graded, just at a reduced value). Ideally, you would have the following source files: `tablegen.c` (the `main` function and the basic GUI), `sort.c` (any code related to sorting the generated table), `io.c` (any code used to read input files or write the final table), and `generate.c` (the code used to generate data for the various rows/columns). Each of these may have an associated header file of the same name (with a .h extension). These .c and .h files represent the only code that you will submit.

IMPORTANT 1: **No error checking** is required on the user's input values. A "real" application would have to do this, of course, but it would add a significant amount of code to this assignment, and we want you to focus on the language itself. So the graders will always enter valid options for columns, row count, and file name.

IMPORTANT 2: All files (.c and .h) must be prepared in the same folder so that they can be compiled from the command line using the following simple gcc command

```
gcc tablegen.c io.c sort.c generate.c
```

You can add the `-Wall -g -gdwarf-4` options during development so that you can test and profile your code.

Note that you may include a README file if some of the application's functionality is not complete. This will allow the grader to give you value for the parts that are working, instead of assuming that nothing is working properly.

Once you are ready to submit, compress all .c/.h/README files (and ONLY these files) into a zip file. The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

Please note that it is your responsibility to check that the proper files have been uploaded. No additional or updated files will be accepted after the deadlines. You can not say that you accidentally submitted an "early version" to Moodle. You are graded only on what you upload.

Good Luck