Project 1a: Sorting Records by Column

You will write a simple sorting program. This program should be invoked as follows:

```
shell% ./varsort -i inputfile -o outputfile [-c column]
```

The above line means the users typed in the name of the sorting program `./varsort` with three arguments:

- an input file to sort called `inputfile`
- an output file called `outputfile` to put the sorted results
- a column number called `column` that we will sort the records based on the data from this column (note that column number starts from 0 as most computer scientists do). This is an optional argument and hence in a bracket. If this optional argument is not provided, then output should be sorted based on column 0.

Input files are generated by a program we give you called genvar.c .

After running `genvar`, you will have a file that needs to be sorted. Note that this file is filled with **binary data** and not ASCII data, so if you try using `cat` or `more` to look at the file, you will see what looks like just a bunch of garbage characters.

The file begins with a **header**. This header is a single integer, **R** containing the number of records in this file.

After the header, the file contains variable-sized records. Each record can be a different size and each is structured as follows:

- **Index:** the first four bytes of each record are an unsigned integer index, starting from **0 to R-1**.
- **Data size:** the second four bytes are an unsigned integer indicating the size N (in integers, or words) of the following data that is associated with this index
- **Data:** N integers (or words) of Data for this index. This data should always stay associated with this index.

For example, one record could look like the following, where each letter represents one byte, and <3> represents the integer (four bytes) holding the number of integers (each of which is 4 bytes) for the associated data:

```
iiii<3>DDDDDDDDDDDD
```

Again, the index is four bytes and this record contains three integers, or 3 * sizeof(integer) = 3 * 4 = 12 bytes.

Note that different records can have indices with different amounts of associated data.

Your goal: to build a sorting program called `varsort` that takes in one of these generated files and sorts it in ascending order (from lowest to highest) based on the data from the specific column. The header and the sorted indices and their associated records are written to the specified output file.

For instance, given records as follows

```
index 0: data_ints: 3 rec: 55 33 77
index 1: data_ints: 4 rec: 35 73 75 53
index 2: data_ints: 2 rec: 73 37
```

Sorting the records by column 0, we should have

```
index 1: data_ints: 4 rec: 35 73 75 53
index 0: data_ints: 3 rec: 55 33 77
index 2: data_ints: 2 rec: 73 37
```

Furthermore, if we sort the records by column 2, we should get

```
index 2: data_ints: 2 rec: 73 37
index 1: data_ints: 4 rec: 35 73 75 53
index 0: data_ints: 3 rec: 55 33 77
```

Because record in index 2 does not have column 2, we then take the **last data** of this record to sort.

Some Details

Using `genvar` is easy. First you compile it as follows:

```
shell% gcc -o genvar genvar.c -Wall -Werror
```

Note: you will also need the header file sort.h to compile this program.

Then you run it:

```
shell% ./genvar -s 0 -n 100 -m 32 -o /tmp/outfile
```

There are four flags to `genvar` . The `-s` flag specifies a random number seed; this allows you to generate different files to test your sort on. The `-n flag` determines how many records to write to the output file. The `-m flag` designates the maximum amount of data that is associated with an index, measured in units of integers (note that records can have LESS than this amount of data). Finally, the `-o` flag determines the output file, which will be the input file for your sort.

The format of the file generated by the `genvar.c` program is very simple: it is in binary form, and consists of the header information and those variable-sized records as described above.

Another useful tool is dumpvar.c . This program can be used to dump the contents of a file generated by `genvar` or by your sorting program.

For example, if run as follows:

```
shell% ./dumpvar -i /tmp/file
```

then `dumpvar` will take the binary data in `/tmp/file` and display the indices and records in a human-readable ASCII format.

When you create files to sort, we strongly recommend that you place those files in `/tmp` rather than in your home directory or course project work space. Files in `/tmp` are stored in the local file system for a particular machine instead of in the AFS distributed file system space; therefore, with `/tmp` you won't run into quota problems and accessing the local files should be relatively quick. Of course, the files you create in `/tmp` on one machine will not exist on a different machine.

You will probably want to the look at the source code for both of these utilities to see how to read

and write to files; in particular, they could be useful for seeing how to understand the variable-sized record format.

A common header file sort.h has the detailed description. There are three different versions of the record that you might find useful in different circumstances:

- `rec_nodata_t`: a structure that contains no data
- `rec_data_t` : a structure that contains the maximum amount of data
- `rec_dataptr_t` : a structure that contains a pointer to the data (allocated elsewhere)

Note that you may NOT simply allocate an array of `rec_data_t` structures for this project because the `rec_data_t` structure assumes the worst-case size for the amount of data that can be associated with an index (`MAX_DATA_INTS`). You must **dynamically** allocate only the amount of memory needed for each record, which might be significantly smaller than the maximum amount. You will want to use `malloc` to allocate the correct amount of data for each record and set the `data_ptr` field in the `rec_dataptr_t` structure to point to this data on the heap.

Hints

In your sorting program, you should just use `open()` , `read()` , `write()` , and `close()` to access files. See the code in `genvar` or `dumpvar` for examples. Make sure you print an appropriate error message and exit correctly if an error occurs!

To sort the data, use any sort that you'd like to use. An easy way to go is to use the library routine `qsort()` . The `qsort()` routine is a bit tricky to use, so you will definitely want to look at the document for it. Typing `man qsort` at the command line will give you a lot of information on how to use the library sorting routine. In particular, you will want to study the example code that demonstrates how to use `qsort` and how to specify a comparison function. Note that the comparison function takes as arguments **pointers** to the objects being compared.

The routine `malloc()` is useful for memory allocation. Make sure to print an error message and exit cleanly if malloc fails!

Remember to write out the header (i.e., the number of records) for your sorted output. `genvar` won't work without it.

For efficiency, you might not want to call `malloc` separately for the data in each record. Maybe you can allocate the memory for all the data at once. If you want to figure out how big the input file is before reading it in, use the `stat()` or `fstat()` calls. After you've read the header and know the number of records in the file, you can calculate how much space is needed for the fixed sized portion of the records (indices, data_ints, and pointers) versus the variable-sized data. After you've allocated enough memory, you can then carefully set up the pointer in the `rec_datptr_t` structure so that record refers to the correct data. Note that when you swap two records during your sort, you will now be swapping the corresponding pointers instead of all the data associated with the records -- which should be much faster for large records.

To exit, call `exit()` with a single argument. This argument to exit() is then available to the user to see if the program returned an error (i.e., return 1 by calling `exit(1)`) or exited cleanly (i.e., returned 0 by calling `exit(0)`).

If you don't know how to use these functions, use the man pages!

To do some preliminary testing of your implementation of `varsort`, test to see if you generate the file sample-out given the input sample-in and column number of 0.

In addition to testing for correct program behavior, we will also be giving points for good programming style and for careful memory management.

As programmers, we often won't be writing all our own code from scratch, and instead will be making contributions inside of an already existing project which other people are also contributing to. In these situations, other programmers will often need to be able to read and understand the code you've written. Because of this, many companies will require its programmers to adhere to a style guideline, so that the task of reading and understanding another person's code is made easier.

For grading on style, we will mostly stick to the style guidelines which Google uses for C++ (more info can be found here) with a few differences specific to C. These differences are in the config file in `~cs537-1/ta/lint/CPPLINT.cfg`. This config file will be used while we run the linter for your code. To grade this, we will be running a kind of program called a "lint program", or a "linter", which just does basic style checking. The linter we will use can be found here:

```
~cs537-1/ta/lint/cpplint.py
```

We will be running it with the following options:

```
cpplint.py --extensions=c,h varsort.c
```

Another important skill to develop in C programming is good memory management. This means freeing any heap space you've allocated when you're done using it! Since memory is a limited resource, we will want to free memory when we're done with it so that it can be reused. To check that your code doesn't contain memory leaks, we will be using a tool known as valgrind. It's a simple tool which moniters every call to malloc (and other memory allocation functions) and free to make sure that all memory allocated to our program is subsequently released when we are done with it. Our tests will be running valgrind on your code in the following way:

```
valgrind --show-reachable=yes ./varsort -i infile -o outfile [-c column]
```

Assumptions and Errors

32-bit integer data: You may assume that the data of the records are **unsigned** 32-bit integers.

Ties in sort: We will not test how you handle the ties in sorting, i.e. it's fine if your sort is unstable.

Data size: You may assume that there are at least one item and no more than USHRT_MAX data items in each record. However, most records may have many fewer data items than the max, so don't allocate this much memory for every record!

Record size: You should be able to handle the file with 0 record, i.e., only a 0 in the file.

File length: May be pretty long! However, there is no need to implement a fancy two-pass sort or anything like that. Moreover, the file will NOT be empty and will always have a header.

Invalid files: If the user specifies an input or output file that you cannot open (for whatever reason), the sort should EXACTLY print: `Error: Cannot open file foo` (with no extra spaces, and assuming the file was named `foo`) and then exit.

Non-negative integer column: You may assume the column in the command argument is

guaranteed to be a valid integer and you can use `atoi()` to convert string to integer. But if you get a negative integer, you should EXACTLY print: `Error: Column should be a non-negative integer` and exit.

Default column: If the column argument is not provided, you should use the default value of 0.

Sorting column may exceed the number of data in some records: If the specified sorting column exceeds the number of data in some record, you should just use the last column of that record to sort.

Too few or many arguments passed to program: If the user runs varsort without enough arguments, or in some other way passes incorrect flags and such to varsort, print `Usage: ./varsort -i inputfile -o outputfile [-c column]` and exit.

Important: On any error code, you should print the error to the screen using `fprintf()` , and send the error message to `stderr` (standard error) and not `stdout` (standard output). This is accomplished in your C code as follows:

```
fprintf(stderr, "whatever the error message is\n");
```

Your grade will primarily depend only on the correctness of your program. However, programs that run signficantly slower than others (i.e., an order of magnitude slower) will be penalized.

Testing

Testing is critical. Testing your code to make sure it works is crucial. Write tests to see if your code handles all the cases you think it should. Be as comprehensive as you can be. After you think you have covered all the edge cases, feel free to test your code with our grading script. To run the script, go to the directory where your `varsort.c` resides and type

```
~cs537-1/ta/tests/1a/runtests
```

We have also provided the log file

```
~cs537-1/ta/tests/1a/runtests.log
```

that passes all the tests for your reference.

General Advice

Start small, and get things working incrementally. For example, first get a program that simply reads in the input file, one line at a time, and prints out what it reads in. Then, slowly add features and test them as you go. Don't worry about performance until you have all of the functionality working correctly.

Keep old versions around. Keep copies of older versions of your program around, as you may introduce bugs and not be able to easily undo them. A simple way to do this is to keep copies around, by explicitly making copies of the file at various points during development. For example, let's say you get a simple version of `varsort.c` working (say, that just reads in the file); type `cp varsort.c varsort.v1.c` to make a copy into the file `varsort.v1.c` . More sophisticated developers use version control systems like CVS or SVN (old days) or mercurial or git (modern times), but we'll not get into that here (though you can, and perhaps should!). **However, you should not upload your code to a public github repo.**