

A8: Indexing Data

[Submit Assignment](#)

Due Sunday by 2:59am **Points** 100 **Submitting** a text entry box or a website url
Available after Mar 8 at 4am


Objectives

- Open, read, manipulate and output files
- Use a hashtable to index data
- Analyze index tradeoffs
- Become familiar with using command line parameters

Summary

In this assignment, we will use a hashtable* to index data for our eventual movie data server. Indexing allows us to more quickly access data. As we saw with caching in class, we like to think about the indexing in order to strike a balance between how much processing we do, how much data we store, and how fast it is to retrieve desired information.

In this assignment, we will read in the data, index it, and then write it out to a file as a report. This has the effect of grouping and sorting data by the specified fields.

* A hashtable is basically a dictionary in Python. [Here is a slide deck](#)  that explains Hashtables in more detail. See the file

```
a8/htll/example_ht.c
```

to see how we create and use this Hashtable implementation.

Sample output

```
adrienne@cs5006-spr2020:~/cs5007$ ./example -g data/test
```

```
indexType: Crime
```

```
4 items
```

```
The Shawshank Redemption
The Godfather
The Godfather: Part II
Pulp Fiction
```

```
indexType: Action
```

```
1 items
```

```
The Dark Knight
```

```
adrienne@cs5006-spr2020:~/cs5007$ ./example -a data/test
```

```
indexType: Uma Thurman
```

```
1 items
```

```
Pulp Fiction
```

```
indexType: Bob Gunton
```

```
1 items
```

```
The Shawshank Redemption
```

```
indexType: Heath Ledger
```

```
1 items
```

```

    The Dark Knight
indexType: Al Pacino
2 items
    The Godfather
    The Godfather: Part II
indexType: Samuel L. Jackson

1 items
    Pulp Fiction
indexType: John Travolta
1 items
    Pulp Fiction
indexType: Robert Duvall

1 items
    The Godfather: Part II
indexType: Marlon Brando
1 items
    The Godfather
indexType: Morgan Freeman
1 items
    The Shawshank Redemption
indexType: Aaron Eckhart

1 items
    The Dark Knight
indexType: Robert De Niro
1 items
    The Godfather: Part II
indexType: Tim Robbins
1 items
    The Shawshank Redemption
indexType: Christian Bale
1 items
    The Dark Knight
indexType: James Caan

1 items
    The Godfather

```

How does this all fit together?

In the bigger scheme of things, we're putting pieces together to build a search engine. A search engine has a few components: a file crawler, a file processor, an indexer, and a query processor. The **file crawler** starts in one place, and traverses to find other files. On the web, this means starting at one page and following all the links on the page; in a file system, it means starting in one directory and traversing through the directory structure until all files have been found. The **file processor** takes each of those files and finds significant words or concepts in the file. The **indexer** populates a data structure that makes it easy to find data records that have similar topics, or find documents that contain a certain word. The **query processor** lets us work with that data structure to get the results based on what we ask for.

File Processing

Now, we're going to read in some movie info.

The program takes in a flag and a filename as arguments, and outputs the results in a given order. The file will be in delimited format, with a pipe (|) as the delimiter. For this assignment, the format is such:

star_rating|title|content_rating|genre|duration|actors_list

- represents a null value. The genre field is a delimited list of genres: it should be parsed, and when indexing via genre, the movie should be considered as all genres.

A few rows:

```
9.2|The Godfather|RI|Crime|175|'Marlon Brando' 'Al Pacino' 'James Caan'
9.1|The Godfather: Part II|RI|Crime|200|'Al Pacino' 'Robert De Niro' 'Robert Duvall'
9|The Dark Knight|PG-13|Action|152|'Christian Bale' 'Heath Ledger' 'Aaron Eckhart'
8.9|Pulp Fiction|RI|Crime|154|'John Travolta' 'Uma Thurman' 'Samuel L. Jackson'
8.9|12 Angry Men|NOT RATED|Drama|96|'Henry Fonda' 'Lee J. Cobb' 'Martin Balsam'
8.9|"The Good, the Bad and the Ugly"|NOT RATED|Western|161|'Clint Eastwood' 'Eli Wallach' 'Lee Van Cleef'
8.9|The Lord of the Rings: The Return of the King|PG-13|Adventure|201|'Elijah Wood' 'Viggo Mortensen' 'Ian McKellen'
```

The program accepts four different flags: `-s`, `-c`, `-g`, `-a`.

- The `-s` flag will index the movies based on star rating.
- The `-c` flag will index the movies based on content rating.
- The `-g` flag will index the movies based on genre.
- The `-a` flag will index the movies based on actor.
- Only one flag may be provided.
- If no flags are set, print out an appropriate message explaining the usage.
- C has a tokenizer functionality (see below). That might help process each line for you.
- If no file is provided, your program should output an appropriate message.

A sample invocation:

```
% ./index_movies -c data/test
Output to file_index.csv
```

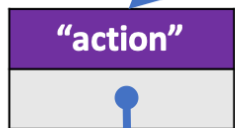
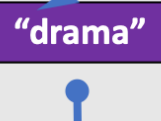
FileParser: reads the file and processes the input
MovieIndex: populates the hashmap w/ movies
Movie: a struct that holds information about a Movie
MovieSet: a set of movies, keeps the description of the set and a linkedlist of movie structs

The program is started for you, with the following structure:

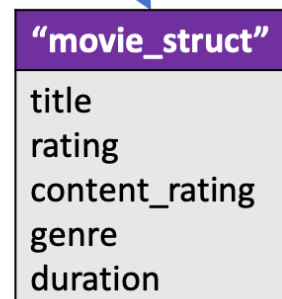
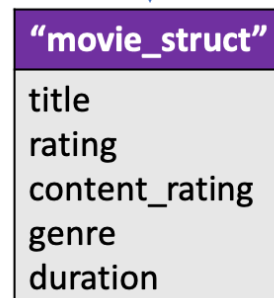
MovieIndex (Hashtable)

Key	Value
"crime"	•
"drama"	•
"action"	•

MovieSet



(LinkedList)



Starting code is in the resources repo, a8 directory.

Hints/Things to Remember

- **getopt** is a useful way to handle command-line arguments.
- **Aspnes (Yale)** explains how command-line arguments work in C.
- Parsing a line of text with delimiters (such as a CSV file, or our file with |) can easily be done by tokenizing; See **strtok_r** for how to do this in C.
- The data file has 1000 lines. It may be helpful to create a small version of this file to work with while you are developing and testing, then move on to the larger file.
 - Some ways to create a smaller file using the command line:
 - `head -n 50 > data_small.txt`
 - `(this is how data/test was created).`
- Make sure your code compiles and runs; even if it crashes or isn't complete. Comment what you need to in order to make sure it compiles.
- Be sure to run the example indexer program, not just the tests.
- Commit early and often.
- Valgrind should report that there are no memory leaks (all heap has been freed) as well as NO ERRORS.
- **Review the grading rubric before getting started.**

How to get started?

Check out the code. We have provided some existing binaries that illustrates (almost) all of the functionality you will be building. Run that a couple of times. Then, read through the code. There are parts marked TODO to indicate where you need to fill in the code. You can run

```
grep TODO *.c
```

to find any outstanding TODOs.

Submission

Submit your assignment by pushing your code to Github and submitting a link to the tag on Canvas. Tag it with the tag `a8-final`.

Optional Challenge

For this assignment, you are provided with an implementation of a Hashtable that follows the same pattern as (and uses!) our Linked List from A7. This allows you to focus on the application part: using a Hashtable to index data.

If you'd like to go a bit deeper, you can implement your own Hashtable. Follow the directions here.

Note: Choosing to implement the Hashtable does NOT impact your grade for this assignment or this class. Align classes tend to have students with a wide range of backgrounds and situations, where some students are able to take on this extra exploration. Implementing a Hashtable is not a primary outcome of the Align program or this class, but can be a valuable experience for the future.

strtok_r()

Just like strtok() function in C, strtok_r() does the same task of parsing a string into a sequence of tokens.

strtok_r() is a **reentrant** version of strtok()

There are two ways we can call strtok_r()

```
// The third argument saveptr is a pointer to a char *  
// variable that is used internally by strtok_r() in  
// order to maintain context between successive calls  
// that parse the same string.  
char *strtok_r(char *str, const char *delim, char **saveptr);
```

Below is a simple C program to show the use of strtok_r() :

```
// C program to demonstrate working of strtok_r()  
// by splitting string based on space character.  
#include <stdio.h>  
#include <string.h>  
  
int main()  
{  
    char str[] = "Geeks for Geeks";  
    char* token;  
    char* rest = str;  
  
    while ((token = strtok_r(rest, " ", &rest)))  
        printf("%s\n", token);  
  
    return (0);  
}
```

Output:

```
Geeks  
for  
Geeks
```

Command line arguments in C/C++

The most important function of C/C++ is `main()` function. It is mostly defined with a return type of `int` and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

To pass command line arguments, we typically define `main()` with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is `int` and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of `argc` would be 2 (one for argument and one for program name)
- The value of `argc` should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If `argc` is greater than zero, the array elements from `argv[0]` to `argv[argc-1]` will contain pointers to strings.
- `argv[0]` is the name of the program , After that till `argv[argc-1]` every element is command -line arguments.

For better understanding run this code on your linux machine.

```
// Name of program mainreturn.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    cout << "You have entered " << argc
          << " arguments:" << "\n";

    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";

    return 0;
}
```

Input:

```
$ g++ mainreturn.cpp -o main
$ ./main geeks for geeks
```

Output:

```
You have entered 4 arguments:
./main
geeks
for
geeks
```

Note : Other platform-dependent formats are also allowed by the C and C++ standards; for example, Unix (though not POSIX.1) and Microsoft Visual C++ have a third argument giving the program's environment, otherwise accessible through `getenv` in `stdlib.h`: Refer [C program to print environment variables](#) for details.

Properties of Command Line Arguments:

1. They are passed to `main()` function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. `argv[argc]` is a NULL pointer.
5. `argv[0]` holds the name of the program.
6. `argv[1]` points to the first command line argument and `argv[n]` points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ' '.

```
// C program to illustrate
// command line arguments
#include<stdio.h>

int main(int argc, char* argv[])
{
    int counter;
    printf("Program Name Is: %s", argv[0]);
    if(argc==1)
        printf("\nNo Extra Command Line Argument Passed Other Than Program Name");
    if(argc>=2)
    {
        printf("\nNumber Of Arguments Passed: %d", argc);
        printf("\n----Following Are The Command Line Arguments Passed----");
        for(counter=0; counter<argc; counter++)
            printf("\nargv[%d]: %s", counter, argv[counter]);
    }
    return 0;
}
```

```
}
```

Output in different scenarios:

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.

```
$ ./a.out
Program Name Is: ./a.out
No Extra Command Line Argument Passed Other Than Program Name
```

2. **Three arguments :** When the above code is compiled and executed with a three arguments, it produces the following output.

```
$ ./a.out First Second Third
Program Name Is: ./a.out
Number Of Arguments Passed: 4
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First
argv[2]: Second
argv[3]: Third
```

3. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
$ ./a.out "First Second Third"
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```

4. **Single argument in quotes separated by space :** When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

```
$ ./a.out 'First Second Third'
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```