



A2

Due Oct 18, 2023 by 4p.m.

Points 10

A2 Instructions CSC209 Fall 2023

Updates and Clarifications

(10/13/2023): The following new starter files for [kar_tree.c](https://q.utoronto.ca/courses/309775/files/28139014?wrap=1) (<https://q.utoronto.ca/courses/309775/files/28139014?wrap=1>)  (https://q.utoronto.ca/courses/309775/files/28139014/download?download_frd=1) and [kar_tree.h](https://q.utoronto.ca/courses/309775/files/28139016?wrap=1) (<https://q.utoronto.ca/courses/309775/files/28139016?wrap=1>)  (https://q.utoronto.ca/courses/309775/files/28139016/download?download_frd=1) now include a ready-made solution for buffered read/write (and a minor bugfix). You don't have to switch to these, but it is one less thing to worry about. Just bear in mind that these files are still missing the solution for `free_tree`, don't accidentally overwrite your solution moving these into your codebase -- not acceptable. If you copy any of this code in, be sure to have all the new contents of `kar_tree.h` in your codebase -- I've heard people encountered a bug from doing `#include <stdio.h>` in `kar_tree.h`. The new starter makes sure this also doesn't happen, take a look.

(10/12/2023): There was a lack of instruction as to what the struct members: `size` and `name` should hold. If you look into the `archive.kar` provided, you'll notice that `size` is used to indicate the number of bytes that make up the file's contents, and that it is unused by directories. You'll also notice that `name` is only the file's name, but that the space for this buffer was at some point used for a longer filepath of some sort -- **ultimately**, the `name` parameter should only be the file/directory's name, and have no other path elements.

Table of Contents

- [1. Introduction](#)
- [2. Submission files](#)
- [3. Feature sets](#)
- [4. kar archive file format](#)
- [5. The “headers”](#)
- [6. Freeing memory](#)
- [7. Error Handling](#)

- [7.1. In general](#)
- [7.2. Creating archives](#)
- [7.3. Extracting archives](#)
- [8. Usage](#)
- [9. File I/O and Helpful system calls](#)
 - [9.1. Write Buffer](#)
 - [9.2. `mkdir` – making directories](#)
 - [9.3. `opendir` – getting a \(DIR *\)](#)
 - [9.3.1. `readdir`](#)
 - [9.4. `struct stat`](#)
- [10. Some notes on when you'll see some of this in class](#)
- [11. Notes on style](#)
- [12. P.S. A Final Note \(perhaps once you're done\)](#)

1. Introduction

Archive managers are an important part of the overall *NIX and computer systems infrastructure.

Most of you have probably encountered `zip`, or perhaps `rar` and `7z` files. They are the modern result of programs that integrate multiple files into a single file known as an *archive*, and (in the case of the examples I mentioned) also compress the resulting file into a smaller form than the starting files. The program known as `tar` can trace its roots back to the original UNIX operating system, and stands for `tape-archive` when magnetic tape (things like cassettes, sometimes in very large rolls) were used to store backups instead of solid-state or spinning plate (hard) drives.

Unlike `zip`, `rar` and `7z` files, `tar` files are not necessarily compressed, but they do allow you to store files and directories (directory trees – as in directories with directories inside them, and so on), while also preserving the usual metadata for files managed by a *NIX system, like permissions, owner and creation dates. Check out the manual for `tar` if you are interested. Recall this means `$ man tar` – as usual.

You will be implementing a less fully featured program called `kar` that:

1. Creates an archive of the specified files/directories
2. Extracts files and directories from an existing kar file into the current working directory (overwriting any existing files with the same names)

The challenge of the assignment will be maintaining a data-structure for multiple possible trees, not only in memory, but to and from a single, linear file.

2. Submission files

Your final submission will consist of the same 6 files you received in your starter code, and nothing more. You should not change the make file called `Makefile` the header `kar_tree.h`. Otherwise, you are free to add as you please to `kar.c`, `kar_tree.c`, `archive.c` and `extract.c`.

Update, commit and push your repository with all of your work contained in the `a2` folder. You should not push any executables (including `.o` files) or testing files.

3. Feature sets

This assignment can be understood through progressively more difficult features that your final program implements. This list is in the order we suggest that you implement things and includes a rough percentage of the overall grade for the assignment:

1. 10% - Program features three commands: `create`, `extract`, and `--help` and a correct `8` message (`--help` or error).
2. 15% - `.kar` archives storing a single file, and recovering the contents of this file e.g.
 1. `$ kar create a2.kar f1.txt` successfully adds `f1.txt` to a (possibly new) archive `a2.kar`
 2. `$ kar extract a2.kar`, if created with the previous example, will create the file `f1.txt` using the archive `a2.kar`, overwriting any file with the same name in the current directory.
3. 10% - `kar` archive with multiple regular (non-directory) files.
 1. `$ kar create a2.kar f1.txt f2.txt` successfully adds `f1.txt` and `f2.txt` to a (possibly new) archive `a2.kar`
 2. `$ kar extract a2.kar`, if created with the previous example, will create the files `f1.txt` and `f2.txt` using the archive `a2.kar`, overwriting any files with the same name in the current directory.
4. 15% - Creating a `kar` archive with a single directory, containing multiple *regular* files.
 1. `$ kar create a2.kar D1/` successfully adds the (non-empty) directory `D1/` to a (possibly new) archive `a2.kar`.
 2. `$ kar extract a2.kar`, if created with the previous example, will create the directory `D1/` using the archive `a2.kar`, overwriting any folder with the same name in the current directory. Note that the files contained within `D1/` should also be extracted.
5. 20% - Creating a `kar` archive with a single directory with files and potential subdirectories (which recursively could contain more subdirectories). Example is the same as previous feature, but may also include directories within `D1/`
6. 10% - Creating a `kar` archive with multiple files and/or directories, such that any directory may have recursive subdirectories.
 1. This is the most general case of the previous two features. Creation might look something like this: `$ kar create a2.kar f1.txt D1/`, where `D1/` contains `f2.txt` and `D2/`, and `D2/` contains `f3.txt`.
 2. `$ kar extract a2.kar`, if created with the previous example, will create the files `f1.txt`, `f2.txt` and `f3.txt` using the archive `a2.kar`. Note that the directories `D1/` and `D2/` should

also be extracted, `f1.txt` should be in the current directory, `f2.txt` should be in `D1/`, and `f3.txt` should be in `D2/`.

Notice that for each stage, we expect that you should be able to archive **and extract the file(s)** stored in the archive. Furthermore, your `kar` program should be able to extract from archives created by another solution that also implements the [file format below](#) and the solution should be able to extract from your archives.

4. kar archive file format

Consider the following directory tree where each file starting with a capital D is a directory, and each file starting with f is a regular file.

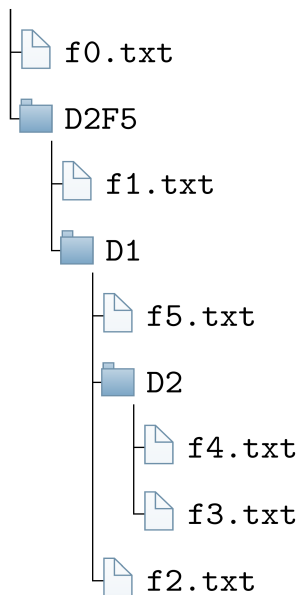
```
[dee@sputnik test]$ ls -R *
f0.txt

D2F5:
f1.txt D1

D2F5/D1:
f5.txt D2 f2.txt

D2F5/D1/D2:
f4.txt f3.txt
```

Here is a better visual. Thanks to Jordan Malek.



Suppose now we executed `kar` as follows within this directory:

```
$ kar create archive.kar f0.txt D2F5/
```

The resulting file's bytes should represent the following structure to be a valid kar archive (we've assumed left-right ordering within each directory has been preserved, this is not necessarily the case, see [5](#)).

Beginning of file (0x0)

Header for f0.txt

All of the contents of f0.txt

Header for D2F5

Header for f1.txt

Contents of f1.txt

Header for D1

Header for f5.txt

Contents of f5.txt

Header for D2

Header for f4.txt

Contents of f4.txt

Header for f3.txt

Contents of f3.txt

Header for f2.txt

Contents of f2.txt

You can find this exact file for you to download [HERE \(https://q.utoronto.ca/courses/309775/files/27853248?wrap=1\)](https://q.utoronto.ca/courses/309775/files/27853248?wrap=1) [↓ \(https://q.utoronto.ca/courses/309775/files/27853248/download?download_frd=1\)](https://q.utoronto.ca/courses/309775/files/27853248/download?download_frd=1)

. Consider using `hexdump -C <archive.kar>` to investigate the contents of your archives.

A more general description of the format is as follows: regular files feature a header immediately followed by their contents. Directories only feature a header, but are immediately followed by the appropriate representations of the files contained within them. This can include directories, which will follow their format recursively i.e. to complete archiving a directory, means it should have a header followed by all of the representations of the files contained within it, before moving on to the next file in the same directory as the original directory (that we started with). Writing recursive information can

be confusing, make sure you understand this format early on! Talk with your peers about it.

5. The “headers”

The headers in each `kar` file are the `structs` created to represent each file in our actual program. In other words, the literal bytes that comprise the data-structures in the program will be stored in the archive file.

The data-structure you will build consists of a linked list to represent all the files a user would like to archive (the command line arguments), and then, for every file that is a directory, **that node** will *separately* also link to another linked list for the contents of that directory... and so on... recursively. This should naturally lend itself to creating the [file format we require](#). The order of the files within each directory will not affect our programs' operation or the correctness overall, so we will just use whatever order the operating system provides the filenames to us in. This means, that the order from the example in [4](#) may be slightly different (files in the same directory, might be in a different order), but the structure of directory headers, followed by the file headers and data for that directory must be maintained.

Each node in our archive tree will represent one file (whether it is a directory or not), and will store some properties of this file, but not the actual contents. The contents are written after a regular file's header, remembering that a directory has no contents in our scheme.

Look at the `arch_tree_node struct`. These structs can be connected to existing linked-lists through their `next_file` pointer member, but also become the parent to a new linked list (that may branch into a tree) when the particular node in question is a directory. You can do this through the `dir_contents` pointer.

6. Freeing memory

Grade percentage: 10% By the end of using kar for any purpose, you will need to free all memory that you have allocated on the heap, anything else will be considered a memory leak and will not result in full marks. Using `valgrind` will allow you to test for this particular problem.

Your main challenge here, will be freeing all the `arch_tree_nodes` successfully, and not losing track of any of them. You might like to review PCRS Week 3: Dynamic Memory Video 5: Nested Data Structures for a general review of what this means.

7. Error Handling

7.1. In general

You should check for errors resulting from system calls, like opening files or any memory you are

allocating. It is sufficient to simply check if the pointer returned by these functions is 0x0, in which case there was an error. Your program should ultimately exit if `malloc` ever fails, but if file I/O fails it depends on which command is being performed. See the example code in `read_node` for how this might look.

7.2. Creating archives

Failing to open a file (e.g. if it doesn't exist) means you should just skip the file. If nothing was ultimately added to the archive, some message (your choice) should be printed to `stderr` and `kar` should exit with a value of 1.

7.3. Extracting archives

For the sake of simplicity we will only test extracting valid archives. A valid archive is one that conforms to the [4](#) and has at least a single file archived within it. Thus, you do not need to handle empty files.

We will assume that you have the appropriate permissions for the current working directory, and do not have to gracefully handle such an error. Feel free to do this if you like, but it is not required.

8. Usage

Your program should print the following usage string if the first argument to the program is `--help`, or there was some sort of error processing arguments – this includes when the command specified was none of the three expected options. This also specifies how your program should behave more generally. You may print this message also at other times you deem reasonable.

```
Usage: kar [COMMAND] [ARCHIVE]
Creates or extracts files from the kar-formatted ARCHIVE.
```

```
COMMAND may be one of the following:
```

```
  create [ARCHIVE] [FILES]
  extract [ARCHIVE]
  --help
```

```
create:
```

```
  Creates an archive with the name specified in [ARCHIVE] consisting of the listed [FILES] which can include directories. Paths to the files are not preserved, each listed file is part of the top-level of the archive. If [ARCHIVE] already exists, it is overwritten. Directories are added recursively, such that all files within the directory are added to the archive.
```

```
extract:
```

```
  Extracts the files from the [ARCHIVE] file into the current directory.
```

```
--help:
```

```
  Prints this message and exits.
```

Note that the indents are constructed using 4 sequential spaces.

9. File I/O and Helpful system calls

Don't forget to close files when you are done with them. If you have an open file pointer called `file_ptr`, it simply means adding `close(file_ptr);`.

9.1. Write Buffer

When archiving and extracting, we will need to be reading from one file and writing to another. We expect you to do this in chunks, so that at the very least, the memory used by your program doesn't fill the entirety of the RAM – if you are archiving a large file. We have provided the constant `WRITE_BUFFER_SIZE` for this exact purpose. You should read from the incoming file *at most* this much, then write what was read to the outgoing file, and then repeat if necessary.

9.2. `mkdir` – making directories

We will use a new system call called `mkdir` available to us if we `#include <sys/stat.h>`. This function is similar to saying `$ mkdir <folder-name>`, but differs insofar as it requires two arguments:

1. The name of the folder (as used at the command line)
2. A `mode_t` structure that describes the permissions for the file

The first argument should be straightforward, and the second can simply be the integer `750`. Recall that permissions can be specified using an numeric format, see `man chmod` for more. Or look into Kerrisk Ch. 18 section 6 for more on `mkdir`.

9.3. `opendir` – getting a (DIR *)

Closing a directory is slightly different than other files, use `closedir` to make sure you've closed it. This system call is largely like `open` for files, but is for directories.

9.3.1. `readdir`

Once you use `opendir(<dir-name>);` to get a `DIR *`. You can use the function `readdir(<open-dir-pointer-variable>)` to get the name of each file in that directory. Each call to `readdir` will return a `dirent *`, yet another type, representing *the next* file in the directory. When `readdir` returns `0x0`, there are no more files to list. One thing you need to watch out for, is that whenever you read from a directory like this, there will be two special files named `'.'` (for a link to the current directory) and `'..'` for a link to the parent directory. You will need to skip these files when making your archive!

You do not *need* to learn anything about `dirent` structs besides that it has a member called `d_name` for the name of each file. You can see much more about this topic in Kerrisk Ch. 18 section 8.

9.4. `struct stat`

We have provided you with part of a function called `create_tree_node` in `archive.c`. This little snippet includes the use of a function called `stat` and it populates a `struct` with information about the open file. Some things you can retrieve from this include the size of the file in bytes through the member `st_size`, and whether or not it is a directory, using `S_ISDIR(st.st_mode);` (assuming your `stat` variable is called `st`).

You can find more information in Kerrisk Ch. 15 section 1.

10. Some notes on when you'll see some of this in class

As you progress through the weeks of the course, topics from class will help you complete more of the assignment.

In week 4, we will be introducing the C `struct`, this will go a long way towards understanding how we will represent and create the linked list. In the subsequent week, we will consider how you might write a `struct` to a file.

11. Notes on style

Total grade percentage: 10%

The style grade is ultimately at the discretion of the TA responsible for grading your work. They will be grading you according to the following principles:

1. Variable names are clear
2. Style such as indentation and bracketing is **consistent** throughout (caveat, the starter code need not change)
3. Functions are never exceedingly long (as in long strings of code, performing **multiple functions** in a script-like fashion)
4. Helper functions are used when needed

Note that comments are not a necessity, but a human will be reading your work.

12. P.S. A Final Note (perhaps once you're done)

Notice how we've managed to really efficiently use the space of our archive in this program. There is minimal wasted space (the filename buffer is just about it). Some food for thought though: how difficult would it be to add a new file to this program? We very well might have to re-write most if not all of the bytes of the archive file, since we've imposed such a strict ordering to where everything goes.

Think a bit about how you might create an archive that would be more amenable to contents that

change. From this point, you aren't very far from creating your own filesystem – where the OS manages the tree structure, and accepts system calls like `fopen` and `mkdir` to update the directory structure. Ideally, the directory tree structure is stored on the memory device (e.g. disk), rather than taking up space in memory (i.e. RAM). Somehow, we would need to be much more flexible about the creation of new files and making good use of space that might have been freed up by deleting other files. Whenever you are reading the `Kerrisk` book, you might now have a better understanding of what motivates a topic that frequently comes up: `Inodes`.

Author: Demetres (dee) Kostas, PhD