Project 1

- **Due Dates**
  - **Project Quiz (https://canvas.umn.edu/courses/354909/quizzes/721050) : 11:59pm, Wednesday 02/01/2023**
  - **Final Submission: 11:59pm, Friday 02/10/2023**
- 8.75% of Course Grade
- Submit to Gradescope
- Projects are to be completed **individually or with a partner. Collaboration outside of your project team is prohibited.** See our **syllabus (https://canvas.umn.edu/courses/354909/assignments/syllabus)** for full academic integrity policies.

Starter Code: proj1-code.zip (https://csci4061-sp23.s3.amazonaws.com/proj01-code.zip)

How to Succeed on This Project

1. **Read the entirety of this specification in detail.** It contains a lot of information that will make the project go more smoothly if you take the time to understand it before starting the project. *Remember that, under our office hours policy, a TA has the right to refuse you assistance if it is clear that you have not read the project spec and other relevant directions.*
2. **Know the purpose of each file in the starter code linked above.** It is worthwhile to take the time to read and understand the code that has been provided to you. Some of the functions provided to you will make your task much easier than it would be otherwise.
3. **Know how to run your code independently of the testing infrastructure we provide.** Test results alone will not be enough to tell you what may be going wrong with your code. You will need to be able to test out your code manually, and *the use of a debugger will be very helpful.* Keep in mind that our tests are not exhaustive – you are expected to implement functionality fo which we have *not* provided tests.
4. **Expect to get stuck, and exercise patience and persistence.** Running into problems that do not have immediately obvious solutions is a normal part of the learning process, particularly with systems programming. Develop and follow a systematic process for tracking down and resolving bugs. **We are happy to help in office hours but will not give out answers. We can only point you in the right direction.**
5. **Start early.** Give yourself time to be stuck, discover insights, and overcome obstacles. It can be hard to think clearly and creatively about solving problems when you feel the pressure of an impending deadline.
6. If you have questions, your best option is to **ask in a public Piazza post.** This gets your question in front of as many people as quickly as possible, and also lets others benefit by seeing the answer to your question.

7. Familiarize yourself with the late submission policy detailed in the **course syllabus (https://canvas.umn.edu/courses/354909/assignments/syllabus)** so that you are not caught off-guard. **No submissions are accepted more than 48 hours after the deadline. Project extensions are not granted except in the case of a validated and acute emergency situation.**

Introduction

This project will help you review C programming, particularly file I/O through the higher-level functions of the `stdio` library. It will also give you a chance to work on a slightly larger and more open-ended C project than you may have seen in CSCI 2021. Finally, this project will serve as an introduction to some of the important principles of systems programming. In particular, we will expect you to be mindful of error checking and failure cases in your code.

The goal of this project is to develop a simplified version of the `tar` utility, called `minitar`. `tar` is one of the popular tools among users of Linux and other Unix-derived operating systems. It creates a single *archive file* from several *member files*. Thus, `tar` is very similar to the `zip` tool we use for code submission in this class, although it does not perform file compression as `zip` does.

The `tar` utility was originally developed back in the era of widespread archival storage on magnetic tapes – `tar` is in fact short for "tape archive." Therefore, archive files created by `tar` have a very simple format that is easy to read and write sequentially. While we won't require you to replicate the full functionality of the original `tar` utility, your `minitar` program will still be quite capable. It will allow you to create to create archives from files, modify existing archives, and extract the original files back from an archive.

`minitar` is fully Posix-compliant, meaning it can freely interoperate with all of the standard tape-archive utility programs like the original `tar`. In other words, `minitar` will be able to read and manipulate archives generated by `tar`, and vice versa. In fact, we will be testing your implementation for equivalence with the a subset of the standard `tar` features.

Note, however, that there are some areas in which the `tar` program installed on the lab machines differs from strict adherence to the Posix standard – meaning the archives produced by `minitar` won't always be byte-for-byte equivalent with those of `tar`.

Grading Criteria

- **Project Quiz (10%)**: We have posted **a quiz (https://canvas.umn.edu/courses/354909/quizzes/721050)** about this assignment description on our class Canvas site. Complete the quiz by the due date. The quiz is to be completed **individually** but multiple attempts are allowed.
- **Automated Testing (50%):** We have provided several tests along with instructions on how to run these tests from the command line. To pass the tests, you will need to ensure that your code

compiles and runs according to the specifications given.

- **Manual Inspection (40%):** Graders will be manually reviewing your code to check for certain features – mainly that you have properly implemented the expected solution and that your code adheres to a reasonable structure and style. **Make sure that you properly check for and deal with errors whenever using a C library function that may indicate an error with its return value.**

Makefile

A `Makefile` is provided as part of this project, much like you have seen for the lab assignments. This file supports the following commands:

- `make`: Compile all code, produce an executable `minitar` program.
- `make clean`: Remove all compiled items. Useful if you want to recompile everything from scratch.
- `make clean-tests`: Remove all files produced during execution of the tests.
- `make zip`: Create a zip file for submission to Gradescope
- `make test`: Run all test cases
- `make test testnum=5`: Run test case #5 only

Automated Tests

Automated tests are included with the project's starter code. **These tests are known to work on CSE labs machines only** but in most cases they should run identically in Linux environments such as the Windows Susbsystem for Linux or a virtual machine.

Don't depend on the test cases and their output for your debugging. Know how to run the `minitar` program directly from the command line, and be ready to use tools like `gdb` to unearth the source of problems you encounter.

Additionally, we have not provided tests for everything you will be expected to implement as part of this project. You will need to do your own testing to verify that your implementation of these features works as expected, and we will be running our own tests against these features as well when grading your submission.

Understanding and Effectively Using the Provided Tests

Unlike on the labs, project tests are slightly more involved. Each test contains a sequence of *steps*, where each step performs some action that is checked for correct output. If a step fails, the rest of the test's steps are skipped.

The automated tests in this project all follow a similar structure, i.e., contain a similar sequence of steps. First, a set of files are copied into the current working directory. This allows your `minitar` implementation to ignore directory structures and do all of its work in the current working directory.

Next, some `minitar` operation is performed (archive create, append, list, update, or extract). All of these operations are detailed below. For every successfully completed operation except *list*, `minitar` is expected to produce no output, just like the real `tar` tool. When an error occurs, `minitar` will need to print out an informative message. You may want to add print statements to help your debugging, but you'll need to remove them to pass the tests.

We often test your `minitar` code for interoperability with the standard `tar` utility. For example, we test archive creation by first building an archive with your `mnitar` code and then trying to extract the files with `tar`. We check that the archive has all the necessary files present **and** that the contents of the file extracted from the archive are identical to the original file included in the provided `test_cases` directory by using the `diff` command.

When test failures occur, they will either be because your archive has too many or too few files (e.g., when we run `tar -xvf` to extract from your archive and it prints out an incorrect list of extracted files) or because the contents of the extracted files do not match the originals (as indicated by the output of the `diff` command).

If a test fails, the files involved in the test (e.g., files intended to be included in an archive) will remain in your project directory in case you would like to inspect them. If an archive file is successfully created before the test fails (named `test.tar` in all the tests), it will also be present in your project directory. You can look at these files in detail to see how they differ from the expected results.

If the test succeeds, a `test_files` directory is set up with all of the files extracted from your archive. You can then look at these files in detail if needed. test.tar is also retained, in your project directory, in case you would like to inspect it.

A good strategy may be to **run a test individually with the `testnum` option, then inspect the contents of `test_files` and `test.tar` to see what went wrong.**

Advice on Manually Testing Your Code

Unlike what you are probably used to in previous classes, **you will need to manually test some of your code** for correctness without explicit verification from automated testing written by course instructors (specifically, you need to test that your program can extract from archives properly). Additionally, we would recommend doing as much manual testing and debugging as possible since it gets you into good habits for this and future courses, and because you may actually find it easier than trying to debug the test cases (which you have less control over).

Helpful Command Line Tools for this Project

You are welcome to attempt to use graphical tools for your own testing, but Linux machines have a variety of utilities installed that make rapid testing easy. Specifically, `xxd` **is useful** for viewing the binary contents of your archive in an (easier) environment. Additionally, remember that the operations you do with your program can also be done directly with the GNU `tar` utility. You can also use a text

editor like `vim` to view the contents of the archive (file structure and file contents) without extracting them. See below for an example of how you might do these things:

```
# These commands let you scroll through the binary content of archives
# Press q to quit, check `less` man page for more details
$ xxd test.tar | less  # view ASCII and hex output
$ xxd -d test.tar | less  # same thing but with decimal offsets

# View archive contents without extracting using Vim (press :q to quit)
$ vim test.tar
```

Manual Grading

We will review the code in the files `minitar_main.c` and `minitar.c` by hand. The purpose of this review is to ensure that you are following good systems programming practices and that your implementation contains the solution elements we expect.

Points are awarded in this category as follows:

Main Function

- **1 point**: Correctly populates linked list with file names from command-line arguments
- **1 point**: Correctly invokes proper archive operations in present of respective flags (`-c` for create, `-a` for append, etc.)
- **1 point**: Prints out file names from within `main()` function for archive list operation
- **4 points**: Properly checks calls to archive functions for errors, cleans up any `file_list_t` instances used (-1 point per missed check or cleanup step)
- **1 point**: Returns proper exit code at end of execution (`0` if success, `1` if error)
- **1 point**: Comments and reasonable code style

Archive Create

- **1 point**: Overwrites existing archive file if already present
- **2 points**: Correctly adds header and contents for each member file to archive (including reading/writing files in chunks)
- **1 point**: Adds two-block footer to archive
- **4 points**: Error Handling and Cleanup. Checks all necessary return values and prints informative message if an error occurs. Frees memory and closes open files. -1 point per missed check or cleanup step.
- **1 point**: Comments and reasonable code style

Archive Append

- **1 point**: Checks that specified archive file already exists and does not overwrite existing archive members

- **2 points**: Correctly adds header and contents for each new member file to archive file (includes reading/writing files in chunks)
- **1 point**: Correctly removes/overwrites old footer and adds new footer at end of archive
- **4 points**: Error Handling and Cleanup. Checks all necessary return values and prints informative message if an error occurs. Frees memory and closes open files. -1 point per missed check or cleanup step.
- **1 point**: Comments and reasonable code style.

Archive List

- **1 point**: Properly reads name from each header block in archive and adds to linked list
- **1 point**: Moves from one header to the next using a seek rather than an inefficient file read.
- **1 point**: Does not print out file names within `get_archive_file_list` function
- **4 points**: Error Handling and Cleanup. Checks all necessary return values and prints informative message if an error occurs, frees memory and closes open files. -1 point per missed check or cleanup step.

Archive Update

- **1 point**: Checks that specified archive file already exists and does not overwrite existing archive members.
- **1point**: Verifies that *all* specified files to update are already present in the archive.
- **1 point**: Correctly adds headers and contents for new member files to archive (includes reading/writing files in chunks)
- **1 point**: Removes/overwrites old footer and adds new footer at end of archive
- **4 points**: Error Handling and Cleanup. Checks all necessary return values and prints informative message if an error occurs. Fress memory and closes open files. -1 point per missed check or cleanup step.
- **1 point**: Comments and reasonable code style.

Archive Extract

- **2 points**: Writes out files of correct names, sizes, and contents (includes reading/writing files in chunks)
- **1 point**: If multiple versions of same file are present, only the most recent version is visible after extraction
- **4 points**: Error Handling and Cleanup. Checks all necessary return values and prints informative message if an error occurs. Frees memory and closes open files. -1 point per missed check or cleanup step.
- **1 point**: Comments and consistent indentation

Starter Code

File	Purpose	Notes
`Makefile`	Testing	Build file to compile and run test cases
`file_list.h`	Provided	Header file for a linked list data structure used to store file names.
`file_list.c`	Provided	Implementation of the linked list data structure for file names.
`minitar.h`	Provided	Header file declaring archive file management functions.
`minitar.c`	*EDIT*	Implementations of functions to perform various archive operations, such as creating archives, updating archives, or extracting data from archives.
`minitar_main.c`	*EDIT*	Implements the command-line interface for the `minitar` application. Parses command-line arguments and invokes archive management functions.
`testius`	Testing	Script to run `minitar` test cases
`test_cases/`	Testing	`minitar` test cases

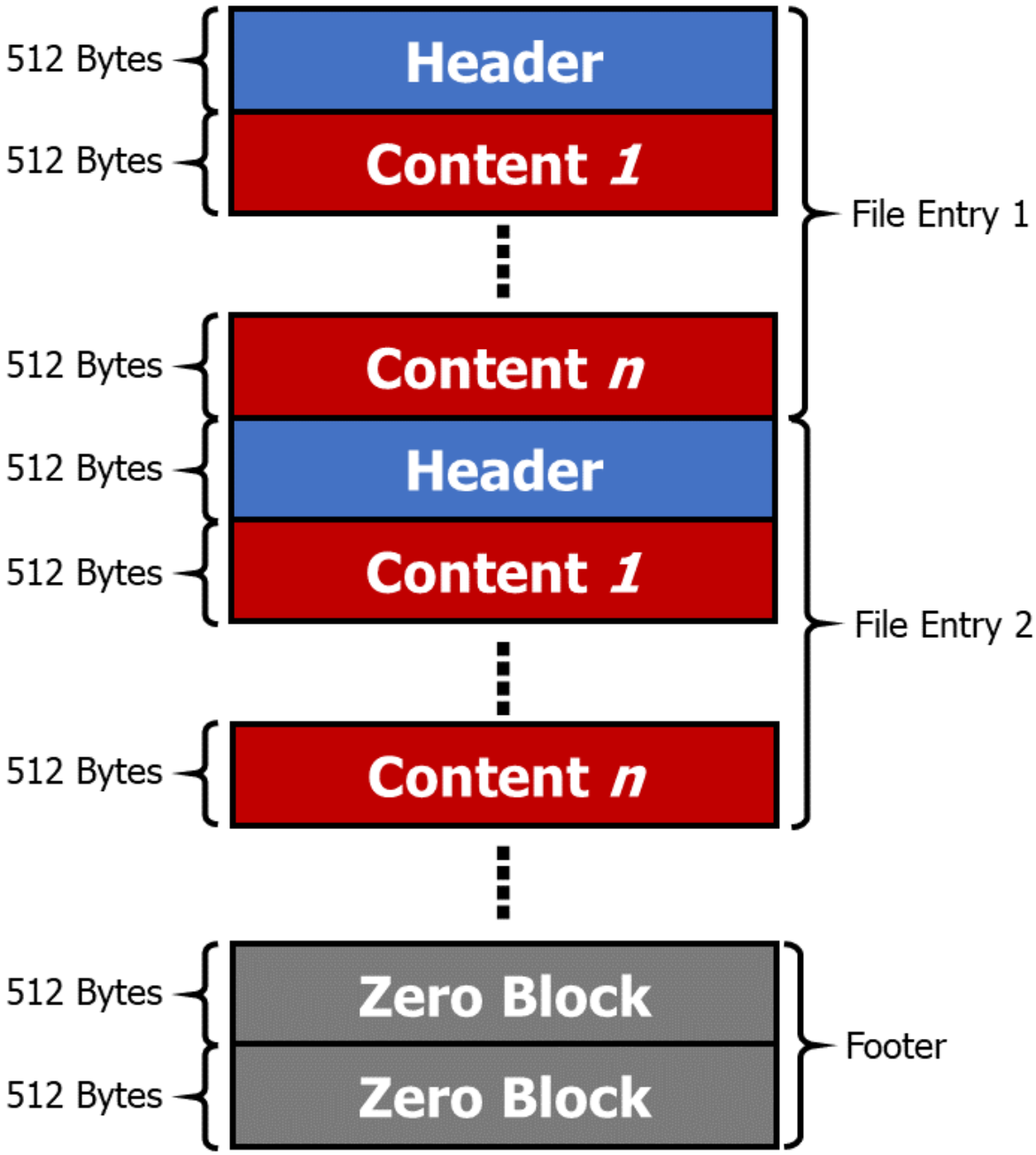
Note that you are only allowed to change the files marked as *EDIT* above. For all other files, the Gradescope autograder will ignore any modifications you make and use the standard version provided with the starter code. Changing other files may result in submission errors, which are not valid grounds for an extension.

Posix-Standard tar Format

The precise structure of tape archive files has been standardized under Posix. This standard enables portability and the exchange of data across different platforms. Any Posix-compliant tape archive program (including `minitar`) can create archives that are understandable to any other compliant program, and vice versa.

`tar` files are made up of a sequence of 512-byte blocks, as shown in the diagram below. Each member file in the archive is represented by a sub-sequence of blocks within the file. The first block in the subsequence is a *header* containing metadata about the file, such as its name and its size in

bytes. The contents of the member file are then reproduced, completely unaltered from the original file, in the subsequent blocks of the archive file.



Note that, because blocks are always 512 bytes in length, and a member file's representation within an archive always uses an integral number of blocks, some space is wasted. If a file is *n* bytes in size, then there will be $\lceil n/512 \rceil$ blocks representing that file in a Posix-compliant tape archive. When a member file's size is not a multiple of 512, its last block will contain trailing zero bytes.

Header Blocks

The structure of a tape archive header is rather involved. We have defined this for you with the `tar_header` type in the `minitar.h` file. The `struct` declaration is reproduced below for your convenience:

```
// Standard tar header layout defined by POSIX
typedef struct {
    // File's name, as a null-terminated string
    char name[100];
    // File's permission bits
    char mode[8];
    // Numerical ID of file's owner, 0-padded octal
    char uid[8];
    // Numerical ID of file's group, 0-padded octal
    char gid[8];
    // Size of file in bytes, 0-padded octal
    char size[12];
    // Modification time of file in Unix epoch time, 0-padded octal
    char mtime[12];
    // Checksum (simple sum) header bytes, 0-padded octal
    char chksum[8];
    // File type (use constants defined in minitar.h)
    char typeflag;
    // Unused for this project
    char linkname[100];
    // Indicates which tar standard we are using
    char magic[6];
    char version[2];
    // Name of file's user, as a null-terminated string
    char uname[32];
    // Name of file's group, as a null-terminated string
    char gname[32];
    // Major device number, 0-padded octal
    char devmajor[8];
    // Minor device number, 0-padded octal
    char devminor[8];
    // String to prepend to file name above, if name is longer than 100 bytes
    char prefix[155];
    // Padding to bring total struct size up to 512 bytes
    char padding[12];
} tar_header;
```

We have provided all of the code you will need to generate a correct tar header from a file, in the `fill_tar_header` function within `minitar.c`. **You are strongly encouraged to read this function to understand how headers are created and to see a good demonstration of proper systems programming practices.** You will also need to access certain header elements, particularly the `name` and `size` fields, when implementing the archive management functions in `minitar.c`.

One peculiarity to watch out for: The Posix tar standard avoids representing numbers in the way we are used to with data types like `int` and `long`. Instead, to avoid the difficulties of different sizes or even endianness for these data types on different platforms, numbers are represented as null-terminated character strings in octal. For example, the `size` field within a header has enough memory to accommodate 11 octal digits and a null terminator. So if a file were to consist of 35 bytes, then the `size` field of its tar header would be `{'0', '0' , '0', '0','0', '0', '0', '0', '0', '4', '3', '\0'}`.

Tar Footer

The end of any tape archive file is marked by a footer: two consecutive blocks of with all zero bytes. These blocks do not contribute any information about the contents of the archive, but they must be

present in all Posix-compliant tape archive files. They make it easy to determine when the end of the archive has been reached while reading its contents sequentially (which you will be doing a lot of with your code).

Implementing the Command-Line Interface

The `tar` program included in most Unix-based environments, like Linux on the CSE Labs machines, has become known for its complicated command-line syntax. We will use a simplified version of the Unix-style command line arguments from the original `tar` for our `minitar` program (as opposed to the "Traditional" or GNU style arguments you may run across in other examples and documentation).

Any command-line invocation of `minitar` will adhere to the following pattern:

```
> ./minitar <operation> -f <archive_name> <file_name_1> <file_name_2> ... <file_name_n>
```

`<operation>` may be any one of the following:

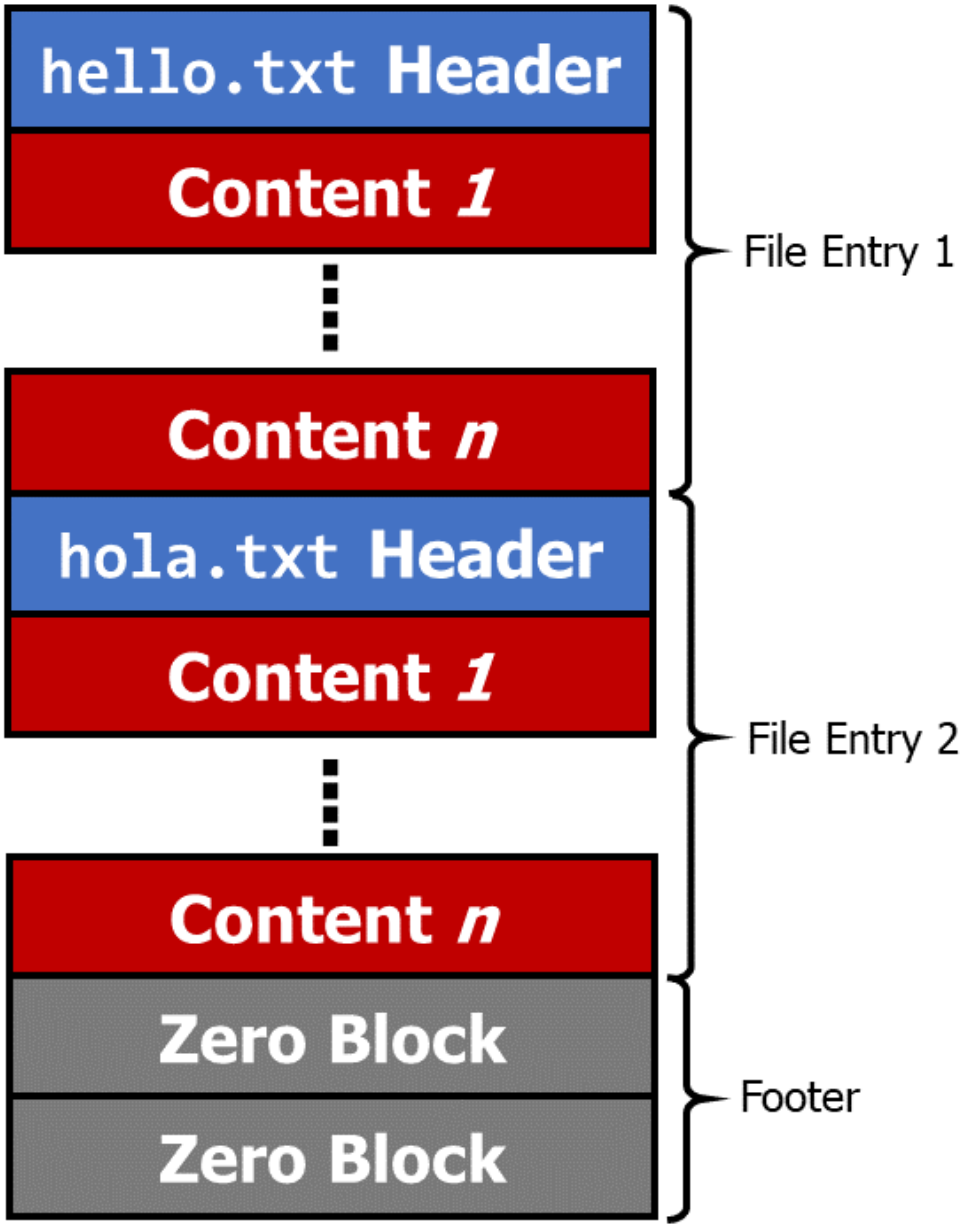
- `-c`: **Create** a new archive file with the name `<archive_name>` and including all member files identified by each `<file_name_i>` command-line argument.
- `-a`: **Append** more member files identified by each `<file_name_i>` argument to the existing archive file identified by `<archive_name>`.
- `-t`: **List** out (print to the terminal) the name of each member file included in the archive identified by `<archive_name>` (no `<file_name_i>` arguments are necessary).
- `-u`: **Update** all member files identified by the `<file_name_i>` arguments contained in the archive file identified by `<archive_name>`. The archive must already contain all of these files, and new versions of each file will be appended to the end of the archive.
- `-x`: **Extract** all member files from the archive identified by the `<archive_name>` argument and save them as regular files in the current working directory. No `<file_name_i>` arguments are necessary.

Implementing the Archive Operations

Archive Create

- **Example Command**: `./minitar -c -f foo.tar hello.txt hola.txt`

You will need to iterate through all specified files, generate a header for each file (check out the `fill_tar_header` function), and write each file's header and contents as a sequence of 512-byte blocks. For example, the tape archive generated by the command above would have the following structure:



Hint: Remember that files are always represented in a tape archive using an integral number of 512-byte blocks. Each file's last block will probably have some unused space.

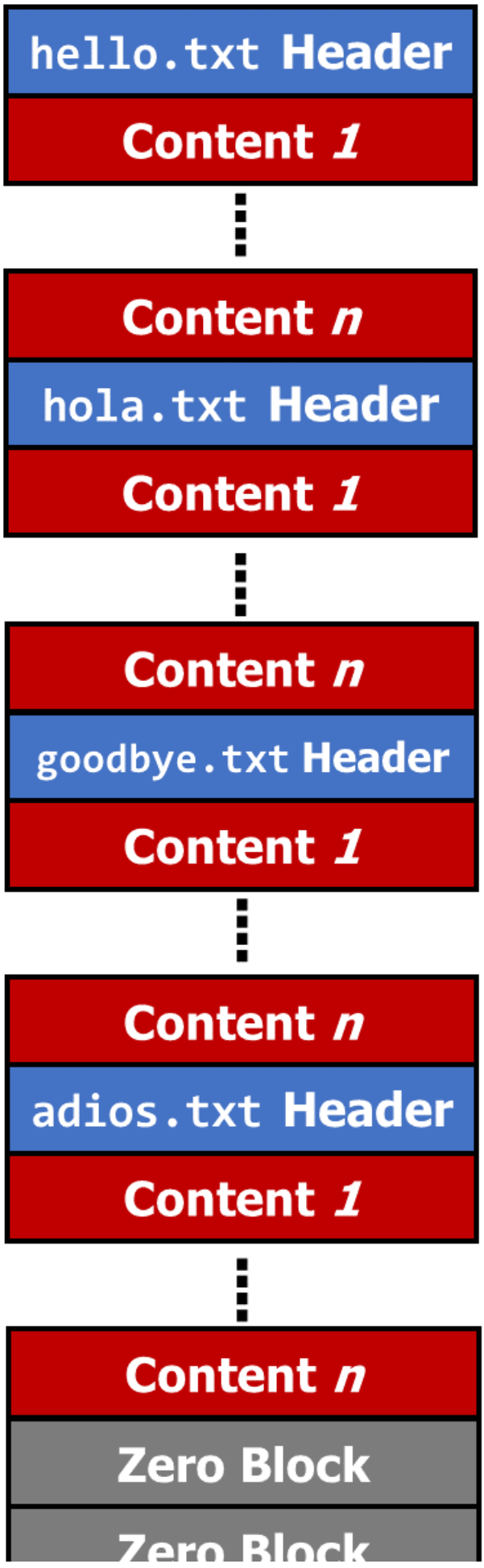
Archive Append

- **Example Command**: `./minitar -a -f foo.tar goodbye.txt adios.txt`

Make sure that the specified archive file (`foo.tar` in the example above) actually exists first. Then, add new representations of the specified member files (`goodbye.txt` and `adios.txt` above) to the archive file.

Hint: Be mindful of the tar footer here. The archive will have an existing footer (two blocks of zero bytes) that you will need to remove before writing blocks for the newly appended files. You may use the `remove_trailing_bytes` function we have provided for this purpose. Then, you'll need to add a footer to the newly extended archive file.

The archive file created after the append command given above would have the following structure:


hello.txt Header
Content 1
Content n
hola.txt Header
Content 1
Content n
goodbye.txt Header
Content 1
Content n
adios.txt Header
Content 1
Content n
Zero Block
Zero Block

List Archive Member Files

- **Example Command**: `./minitar -t -f foo.tar`

Iterate through the archive file's header blocks, printing out the `name` field from each one.

Hint: Think carefully about how you'll know when you've reached the end of the archive and can stop reading more data from the file.

Using the running example archive from above, the output of the list operation would be:

```
hello.txt
hola.txt
goodbye.txt
adios.txt
```

Update Archive Member Files

- **Example Command**: `./minitar -u -f foo.tar goodbye.txt adios.txt`

A user may want to include a newer version of one or more member file already present in an archive. This is effectively the same as an append -- the contents added to the archive are just new contents for a previously added file rather than a brand new file. This means a single archive can have two (or more) sequences of blocks for a file with the same name.

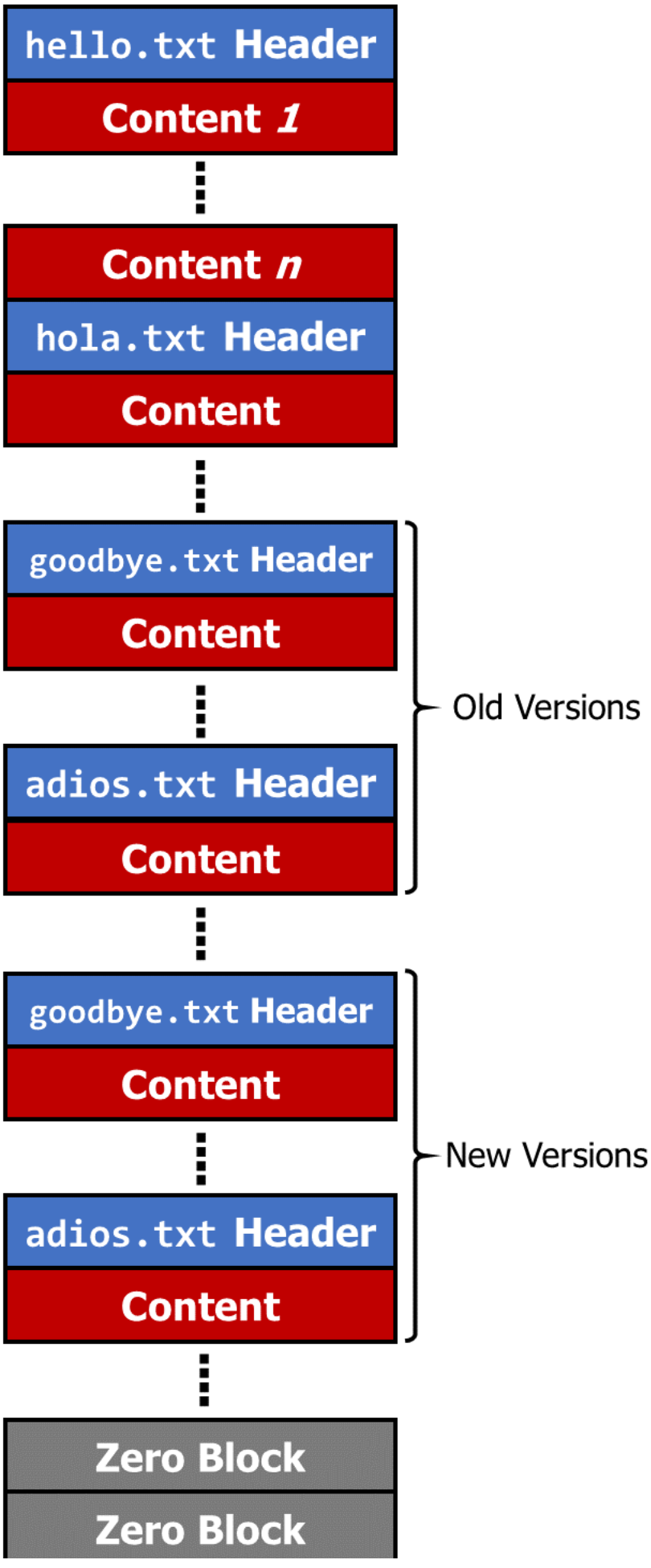
This can lead to strange things like the same file name getting printed out twice in a list operation. However, this situation can also nicely simplify your implementation, as update operations are very similar to append operations.

First, verify that all of the files are present in the specified archive file.

If the user attempts to update a file that does not have a previous version already stored in the specified archive file (e.g., `foo.tar` does not contain a version of `adios.txt` in the example command above), then your program should print out the following message: `Error: One or more of the specified files is not already present in archive`. Note that you will need to match this error message exactly to pass some of our tests.

- **Hint**: This is essentially just an append operation with the additional step of verifying that each file already has a version present in the archive. Try to reuse your code from the append operation as much as possible.

Using our running example, the archive would have the following structure after the update command given above.


hello.txt Header
Content 1
Content n
hola.txt Header
Content
goodbye.txt Header
Content
adios.txt Header
Content
Old Versions
goodbye.txt Header
Content
adios.txt Header
Content
New Versions
Zero Block
Zero Block

Extract Member Files From Archive

- **Example Command**: `./minitar -x -f foo.tar`

Iterate through the archive and create a new file in the current working directory for each member file contained in the archive. If there are multiple versions of the same file present in the archive, only the version of the file most recently added to the archive should be extracted.

- **Hint**: Be careful about sizes here. The contents of all member files in the archive are represented as 512-byte blocks, but you won't want to write out the full contents of the file's last block if that file's size is not a clean multiple of 512.

We have not provided automated tests for archive extraction, although we will still be testing your submission for this functionality. You will need to test extraction yourself. In particular, make sure your implementation is able to correctly extract files from an archive generated by `minitar` or an archive generated by the standard `tar` program.

Simplifying Assumptions and Hints

Don't worry about directory structure for your archives. The real `tar` tool allows the user to archive files living in many different directories at once and then reproduces this directory structure upon extraction. For `minitar`, you can assume any file added to an archive is located in the current working directory and you can simply extract all files in the archive to the current directory.

Don't forget to check return values of all library functions! I/O functions, in particular, usually can indicate errors with their respective return values. We *will* be assessing your consistency in checking for and handling errors in our manual grading. Look at the starter code we have provided to see examples of good systems programming practice.

Use the documentation provided by `man` to learn your way around helpful C functions.

Remember to be think carefully about the contents of memory. The `memset()` function will come in handy in multiple places in this project.

Study up on use of the `stdio` functions. All of the following functions will be useful in this project.

- `fread()` and `fwrite()` to read and write data
- `ftell()` and `fseek()` to get and set the current read/write position in a file
- Look at the variants of `scanf()` when you need to parse the `size` field from a header block (remember, it's in a zero-padded octal representation)

Remember, many C library functions set the `errno` global variable to indicate the reason for a failure. Use the helpful `perror()` function to print out a useful message based on the value of `errno`.

Avoid using the low-level C I/O library functions using file descriptor `int` values like `open()`. Use functions like `fopen()` instead. We'll learn all about low-level I/O later, but you won't need it for this project. **Do not mix high-level (e.g., `fopen()`) and low-level (e.g., `open()`) functions when working on the same file.** This can lead to confusing results that won't make sense until we learn about buffering.

Leverage the starter code as much as possible. Use the provided `file_list_t` data structure wherever you can, and use our functions like `remove_trailing_bytes()` and `fill_tar_header()` to save yourself implementation effort.

Reuse your own code as much as possible. The archive operations share a lot of their logic, such as iterating over 512-byte blocks, appending blocks for new files, or getting a list of names for all of an archive's member files. Write helper functions and reuse them multiple archive operations where possible.

Why is there no dedicated function to update an archive like there is to create, append, extract, etc?

As stated above, an update is very similar to an append. Reuse your code for an append operation to implement the update operation.

How can I look at the archive files my code produces to see what is wrong?

The `xxd` command converts binary files to hex for easier reading by humans. So you can do something like `xxd test.tar > test.hex` and then look over the `test.hex` file. Look to see if the contents of your file adhere to the expected structure. Is there a correct header for each file? Are file contents in 512-byte blocks? Does your archive have the two required zero blocks at its end?

Submission

You must submit your code to Gradescope to receive credit for this project. You should run the `make zip` command to conveniently create a zip archive suitable for uploading to Gradescope.

Remember, you can only modify the files designated as *EDIT* in the table above. Modifying other files will cause your code to fail the autograder tests.

It is your responsibility to ensure that the Gradescope autograder results match your expectations based on local testing. The autograder is the final authority on the automated testing portion of your grade.

You may also wish to review the late submission policy detailed in the **course syllabus (https://canvas.umn.edu/courses/354909/assignments/syllabus)** to understand what score you will receive if submitting your work after the deadline. **No project submissions are accepted more than 48 hours after the deadline.**