

ARVİK



Fundamentally, the **arvik-md4** assignment is about **reading and writing files**. That's really it.

- There are no complex data structures you need to implement.
- You don't **have** to call `malloc()` and manage memory.
- There are no really messy strings to parse, mostly.
- Just read files and write files.

Binary Files

To be sure, the **archive member files** you read and write **may be binary files**, so you must **treat them all as binary files**.

If a call to **fopen()** occurs anywhere in your code, **you are doing it wrong!** You need to use **open()**, not `fopen()` for the files you read or write.

Likewise, you **must not use printf() or fprintf() on any of the archive files you create**. You will use the **write()** function to add data into an archive file.

You will use **read()** to fetch data from a file.

Non-file Output

When you display **diagnostic messages** (error statements, debugging messages, or verbose output) calling `fprintf(stderr, ...)` is fine. **In fact**, you must send all diagnostic messages to `stderr`, not `stdout`.

Do not comingle output with diagnostic messages.

sizeof()

Technically, **sizeof()** is an operator, not a function. However, we often describe it as a function and certainly use it like a function (with parenthesis following it). The following statements in C are equivalent:

```
size_t siz1 = sizeof(int);  
size_t siz1 = sizeof int;
```

We use the parenthesis as grouping for the operand to the **sizeof()** operator. **I always use parenthesis with sizeof().**

There is not a `man` page for **sizeof()**, in the same way that there is not a `man` page for **+** operator.

More on `sizeof()`

The `sizeof()` operator returns the **number of bytes** required to represent a given datatype. We can use either the name of a datatype or an instance of a datatype as the operand to `sizeof()`.

```
char c;  
char *cp;  
int i;  
int *ip;
```

I have a preference to use a datatype as the operand to `sizeof()`, rather than an instance of a datatype. It is not not always possible though.

```
sizeof(char);           // Guaranteed to be 1.  
sizeof(c);  
sizeof(int)             // System dependent, but 4 for us.  
sizeof(i);  
  
sizeof(char *);         // All pointers are the same size.  
sizeof(cp);             // On our system, pointers are  
sizeof(int *);          // 8 bytes (64 bits).  
sizeof(ip);
```

strlen()

Without any doubt, `strlen()` is a function call. It is part of the C-strings library (in `string.h`).

Quoting directly from the `man` page:

The `strlen()` function calculates the **length of the string** pointed to by `s`, **excluding the terminating null byte** (`'\0'`).

Like `sizeof()`, the `strlen()` function returns a `size_t` type (an unsigned long int). We rarely think of things having negative size or negative length. Maybe in some weird physics...

`strlen()` vs `sizeof()`

When you want to know the **number of bytes required for a datatype**, you use `sizeof()`.

When you want to know the **length of a string**, you use `strlen()`.

If you have code that looks like this:

```
size_t siz = sizeof("abc");
```

What is the result?

What is the type of "abc"?

It is a `char []`. An **array of char**, with **4 elements**. Sooo, the value of `siz` is 4, the number of bytes required represent the 3 characters **plus** the NULL.

strlen() vs sizeof()

If you have code that looks like this:

```
size_t siz = sizeof("supercalifragilisticexpialidocious");
```

What is the result?

What is the type of "supercalifragilisticexpialidocious"? It is also an **array of char**. So, the value of `siz` is **35**, the number of bytes required represent the **ALL characters plus the NULL**.



This would give a different result:

```
size_t siz2 = strlen("supercalifragilisticexpialidocious");
```

Specifically, it would be **34**.

`strlen()` vs `sizeof()`

If you have code that looks like this:

```
char *str = {"abc"};  
size_t siz = sizeof(str);
```

What is the result?

What is the type of `str`?

This time, `str` is a `char *`, **not** an array of `char`.

So, `siz` is equal to **8**, the size of all pointers on our system.

This would give a different result:

```
size_t siz2 = strlen(str);
```

Specifically, it would still be **3**.

strlen() vs sizeof()

```
{
    size_t siz = sizeof("abc");

    printf("sizeof(\"abc\") = %zu\n", siz);
    printf("strlen(\"abc\") = %zu\n", strlen("abc"));
}

{
    char arr[] = {"abc"};
    size_t siz = sizeof(arr);

    printf("sizeof(arr[]) = %zu\n", siz);
    printf("strlen(arr) = %zu\n", strlen(arr));
}

{
    size_t siz = sizeof(char [4]);

    printf("sizeof(char [4]) = %zu\n", siz);
}
```

The diagram illustrates the behavior of `sizeof` and `strlen` for different array declarations. It consists of three code blocks, each enclosed in curly braces. The first block shows `sizeof("abc")` and `strlen("abc")`. A green arrow labeled '4' points to `sizeof("abc")`, and a blue arrow labeled '3' points to `strlen("abc")`. The second block shows `sizeof(arr)` and `strlen(arr)` for a character array `arr` containing "abc". A green arrow labeled '4' points to `sizeof(arr)`, and a blue arrow labeled '3' points to `strlen(arr)`. The third block shows `sizeof(char [4])` and `printf("sizeof(char [4]) = %zu\n", siz);`. A green arrow labeled '4' points to `sizeof(char [4])`.

strlen() vs sizeof()



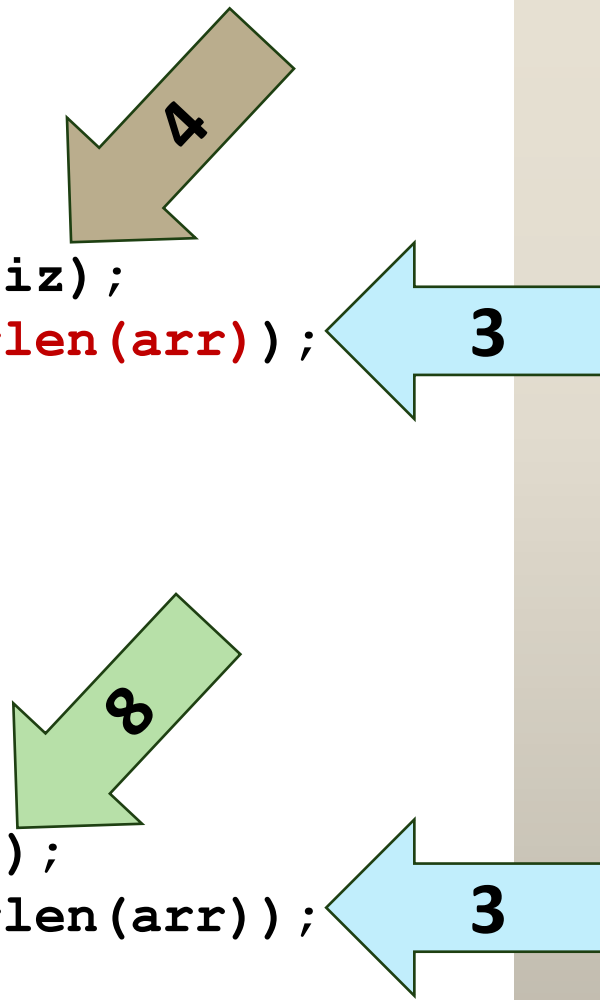
```
{  
    char *arr = {"abc"};  
    size_t siz = sizeof(arr);  
  
    printf("sizeof(arr) = %zu\n", siz);  
    printf("strlen(arr) = %zu\n", strlen(arr));  
}  
  
{  
    size_t siz = sizeof(char *);  
  
    printf("sizeof(char *) = %zu\n", siz);  
}
```

Diagram illustrating the output of the code snippets:

- For the first snippet, `sizeof(arr)` is 8 (indicated by a green arrow pointing to the first `printf` statement) and `strlen(arr)` is 3 (indicated by a blue arrow pointing to the second `printf` statement).
- For the second snippet, `sizeof(char *)` is 8 (indicated by a green arrow pointing to the `printf` statement).

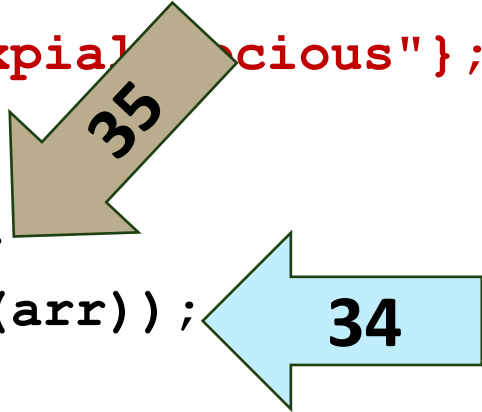
strlen() vs sizeof()

```
{  
    char arr[] = {"abc"};  
    size_t siz = sizeof(arr);  
  
    printf("sizeof(arr[]) = %zu\n", siz);  
    printf("strlen(arr) = %zu\n", strlen(arr));  
}  
  
{  
    char *arr = {"abc"};  
    size_t siz = sizeof(arr);  
  
    printf("sizeof(arr) = %zu\n", siz);  
    printf("strlen(arr) = %zu\n", strlen(arr));  
}
```



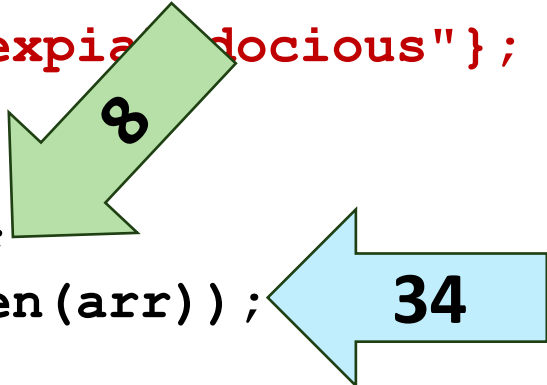
strlen() vs sizeof()

```
{  
    char arr[] = {"supercalifragilisticexpialidocious"};  
    size_t siz = sizeof(arr);  
  
    printf("sizeof(arr[]) = %zu\n", siz);  
    printf("strlen(arr) = %zu\n", strlen(arr));  
}
```



A diagram illustrating the difference between `sizeof` and `strlen` for a character array. A brown arrow points from the `sizeof(arr)` expression to the value 35, representing the total size of the array in bytes, including the null terminator. A light blue arrow points from the `strlen(arr)` expression to the value 34, representing the length of the string in characters, excluding the null terminator.

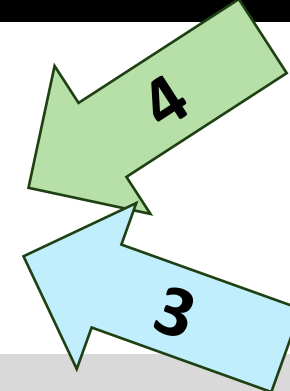
```
{  
    char *arr = {"supercalifragilisticexpialidocious"};  
    size_t siz = sizeof(arr);  
  
    printf("sizeof(arr) = %zu\n", siz);  
    printf("strlen(arr) = %zu\n", strlen(arr));  
}
```



A diagram illustrating the difference between `sizeof` and `strlen` for a pointer to a character array. A green arrow points from the `sizeof(arr)` expression to the value 8, representing the size of the pointer variable in bytes. A light blue arrow points from the `strlen(arr)` expression to the value 34, representing the length of the string in characters, excluding the null terminator.

Final Note

```
{  
    char arr1[] = {"abc"};  
    size_t siz1 = sizeof(arr1);  
    char arr2[] = {'a', 'b', 'c'};  
    size_t siz2 = sizeof(arr2);  
  
    printf("sizeof(arr1[]) = %zu\n", siz1);  
    printf("sizeof(arr2[]) = %zu\n", siz2);  
}
```



Note the difference between `arr1` and `arr2`.

- The **`sizeof(arr1)`** will be 4.
- The **`sizeof(arr2)`** will be 3.
- A **`strlen(arr1)`** will work.
- A **`strlen(arr2)`** will likely result in a seg-fault.

Macro Reticence

```
#define OPTIONS "cxtvVf:h"

#define ARVIK_FILE "#<arvik4>\n"

#define ARVIK_NAME_LEN 30
... ..

typedef struct arvik_header_s {
    char arvik_name[ARVIK_NAME_LEN]; // Member file name, sometimes < terminated.
    char arvik_date[ARVIK_DATE_LEN]; // File date, decimal seconds since Epoch.
    char arvik_uid[ARVIK_UID_LEN]; // User ID, in ASCII decimal.
    char arvik_gid[ARVIK_GID_LEN]; // Group ID, in ASCII decimal.
    char arvik_mode[ARVIK_MODE_LEN]; // File mode, in ASCII octal.
    char arvik_size[ARVIK_SIZE_LEN]; // File size, in ASCII decimal.
    char arvik_term[ARVIK_TERM_LEN]; // Always contains ARVIK_TERM.
} arvik_header_t;
```

const vs Macro

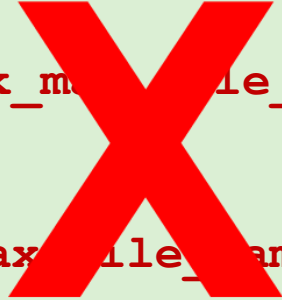
```
static const char arvik_options[] = {"cxtvVf:h"};

static const char *arvik_file_tag = {"#<arvik>\n"};

static const unsigned short arvik_max_file_name_length = 30;

typedef struct arvik_header_s {
    char    arvik_name[arvik_max_file_name_length];

    ... ..
} arvik_header_t;
```



ERROR

Macro Wins!

```
static const char arvik_options[] = {"cxtvVf:h"};

static const char *arvik_file_tag = {"#<arvik>\n"};

#define ARVIK_NAME_LEN 30
static const unsigned short arvik_max_file_name_length = ARVIK_NAME_LEN;

typedef struct arvik_header_s {
    char        arvik_name[ARVIK_NAME_LEN];

    ... ..
} arvik_header_t;
```

The Macro is There to Help

```
#define OPTIONS "xctTf:hv"

#define ARVIK_FILE "#<arvik the archive champion>\n"

#define ARVIK_NAME_LEN 27

typedef struct arvik_header_s {
    char      arvik_name[ARVIK_NAME_LEN];

    ... ..
} arvik_header_t;
```



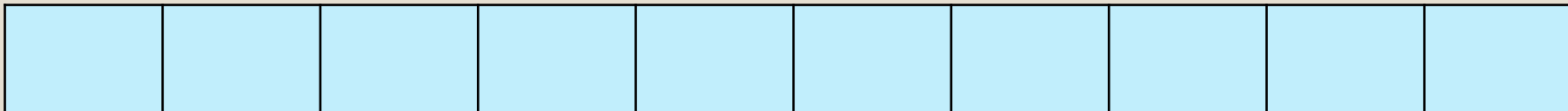
Consult the man page



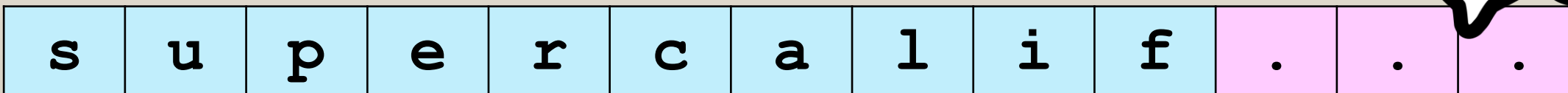
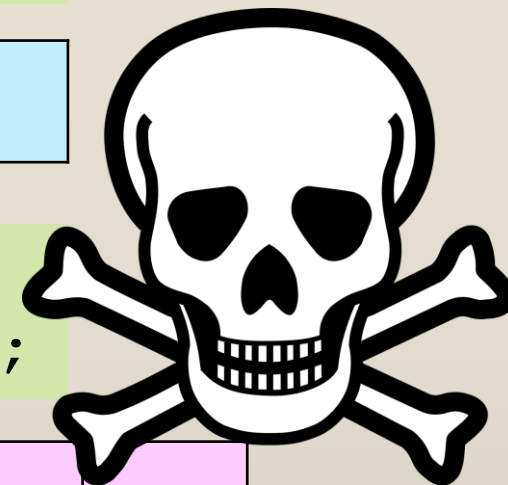
From the man page for `strncpy()`:

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. **Warning: If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.**

```
#define NAME_LEN 10  
char file_name[NAME_LEN] = {'\0'};
```



```
strcpy(file_name  
       , "supercalifragilisticexpialidocious");
```



From the **strcpy** man page:

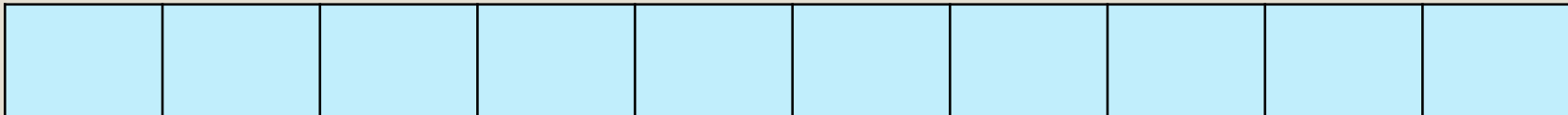
Beware of buffer overruns! (See BUGS.)

BUGS

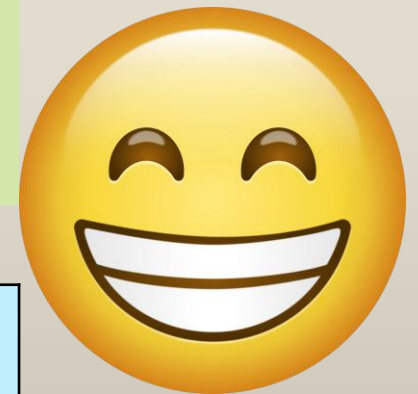
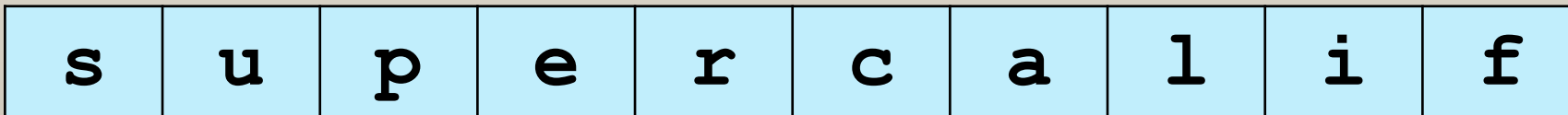
If the destination string of a **strcpy()** is not large enough, then anything might happen.

Buffer overflow!!!

```
#define NAME_LEN 10  
char file_name[NAME_LEN] = {'\0'};
```



```
strcpy(file_name  
      , "supercalifragilisticexpialidocious"  
      , NAME_LEN);
```



No buffer overflow!!!
But, also, not NULL terminated!!!

```
# define ARVIK_TAG    "#<arvik4>\n"    // String that begins an arvik file.
# define ARVIK_TERM   "$\n"            // String in end of each arvik header and footer.
# define ARVIK_NAME_TERM '<'          // Char at end of file name.

typedef struct arvik_header_s {
typedef struct arvik_header_s {
    char arvik_name[ARVIK_NAME_LEN];    // Member file name, sometimes < terminated.
    char arvik_date[ARVIK_DATE_LEN];    // File date, decimal seconds since Epoch.
    char arvik_uid[ARVIK_UID_LEN];      // User ID, in ASCII decimal.
    char arvik_gid[ARVIK_GID_LEN];      // Group ID, in ASCII decimal.
    char arvik_mode[ARVIK_MODE_LEN];    // File mode, in ASCII octal.
    char arvik_size[ARVIK_SIZE_LEN];    // File size, in ASCII decimal.
    char arvik_term[ARVIK_TERM_LEN];    // Always contains ARVIK_TERM.
} arvik_header_t;
} arvik_header_t;

typedef struct arvik_footer_s {
    char md4sum_header[MD4_DIGEST_LENGTH * 2];
    char md4sum_data[MD4_DIGEST_LENGTH * 2];
    char arvik_term[ARVIK_TERM_LEN];    // Always contains ARVIK_TERM.
} arvik_footer_t;
```

Just **cat**s the **arvik** file to the terminal. It does not look scary.

```
a8f8757d92c867db5d2a1a57b7d96a37bb1a936595d591b5f9a51f8791123712$  
2-s.txt<                                -107617704            18781 10265 100456    82           $  
2222222222222222222222222222222222222222222222222222222  
2222222222222222222222222222222222222222222222222222222
```

The arvik header **metadata**.

~~#<arvik4>~~

111

a8f8757d92c867db5d2a1a57b7d96a37bb1a936595d591b5f9a51f8791123712\$

The arvik file data.

The arvik footer MD4 checksums.

Each data section is 2 byte aligned. If it would end on an odd offset, a newline ('\n', 0x0A) is used as filler. The size of the data is not changed.

Valid arvik Files

#<arvik4>

#<arvik4>

File header

File data

File footer

#<arvik4>

File header

File data

File footer

File header

File data

File footer

#<arvik4>

File header

File data

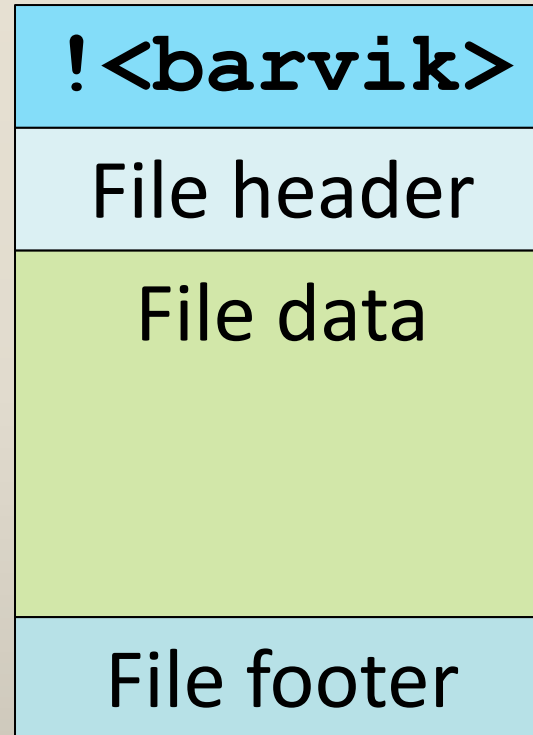
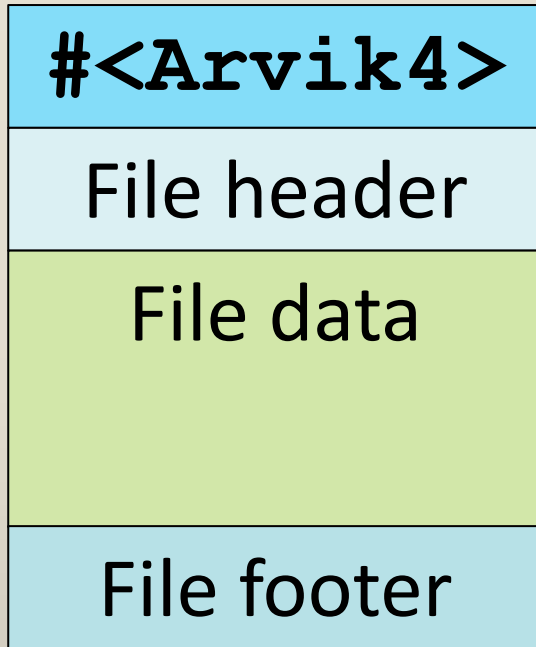
File footer

File header

File data

File footer

Baaad Files



An arvik File with Only a Tag

```
./arvik-md4 -c > 0-archive-members.arvik
```

```
#<arvik4>
```

How many characters are in the ARVIK_TAG macro?

```
# ls -l 0-archive-members.arvik
-rw----- 1 rchaney them 10 Oct 30 09:53 0-archive-members.arvik
```

From arvik.h

```
#define ARVIK_TAG    "#<arvik4>\n"    // String that begins an archive file.
```

An arvik File with a **Zero** Size File

```
./arvik-md4 -c 0-s.txt > 1-archive-member-zero-bytes.arvik
```

#<arvik4>
File header
File footer

10 bytes are used for the tag at the beginning of the file.

78 bytes are used for the header structure.

66 bytes are used for the footer structure.

```
# ls -l 1-archive-member-zero-bytes.arvik
-rw-rw-r-- 1 rchaney them 154 Apr 7 20:29
1-archive-member-zero-bytes.arvik
```

```
sizeof(arvik_header_t) == 78
sizeof(arvik_footer_t) == 66
```

An arvik File with 1 Archive Member

```
./arvik-md4 -c 1-s.txt -f 1-archive-member-41-bytes.arvik
```

#<arvik4>

File header

File data

File footer

10 bytes are used for the tag at the beginning of the file.

78 bytes are used for the header structure.

The 1-s.txt file contains exactly 41 bytes

66 bytes are used for the header structure.

```
# ls -l 1-archive-member-41-bytes.arvik
-rw-rw-r-- 1 rchaney them 196 Feb  4 10:25
                                1-archive-member-41-bytes.arvik
```

$10 + 78 + 41 + 66 + 1 = 196$

Each data section is 2 byte aligned. If it would end on an odd offset, a newline ('\n', 0x0A) is used as filler. The size of the data is not changed.

An arvik File with 3 Archive Members

```
./arvik-md4 -c [1-3]-s.txt -f 3-archive-members.arvik
```

#<arvik4>	9 bytes
File1 header	64 bytes
File1 data	41 bytes
File1 footer	12 bytes
File2 header	64 bytes
File2 data	82 bytes
File2 footer	12 bytes
File3 header	64 bytes
File3 data	123 bytes
File3 footer	12 bytes

```
# ls -l [1-3]-s.txt 3*.arvik
 41 Jul 18 1975 1-s.txt
 82 Aug 4 1966 2-s.txt
123 Aug 4 1966 3-s.txt
485 Apr 8 05:32 3-archive-members.arvik
```

$$10 + (3 * 78) + (41 + 1) + 82 + (123 + 1) + (3 * 66) = 690$$

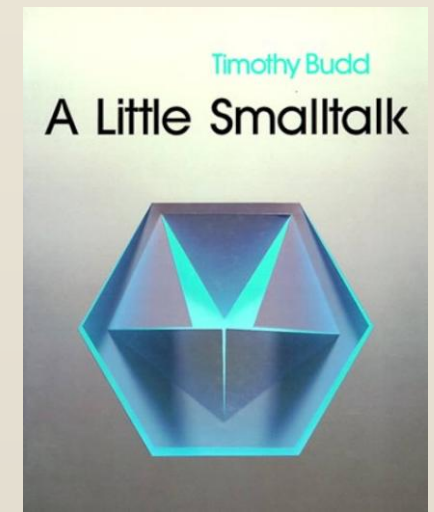
Each data section is 2 byte aligned. If it would end on an odd offset, a newline ('\n', 0x0A) is used as filler. The size of the data is not changed.

Accessing Files in argv[]

From `~rchaney/Classes/cs333/src/argc_argv/getopt_short.c`

```
////////////////////////////////////  
// A this point, getopt() has processed all of the options on the command  
// line, but that may not be everything that is on the command line. There  
// could be additional items on the command line that are not used as  
// switches. An example of when something like this can happen is with  
// the ls command:  
//     ls -l argv.c environ.c envp.c getopt_short.c  
// Everything following the -l command line option are still contents of  
// argv, but getopt() won't process them.  
... ..  
if (optind < argc)  
{  
    fprintf(stderr, "\nThis is what remains on the command line:\n");  
    for(int j = optind; j < argc; j++)  
    {  
        // If you want to do something with these values on the command line  
        // you'll need to use something other than getopt().  
        fprintf(stderr, "\t%s\n", argv[j]);  
    }  
}
```

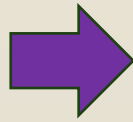
A Little Small TOC



#<arvik4>
File1 header
File1 data
File1 footer
File2 header
File2 data
File2 footer
File3 header
File3 data
File3 footer

```
# ./arvik-md4 -t -f 3-archive-members.arvik  
1-s.txt  
2-s.txt  
3-s.txt
```

A Little Small TOC - 2

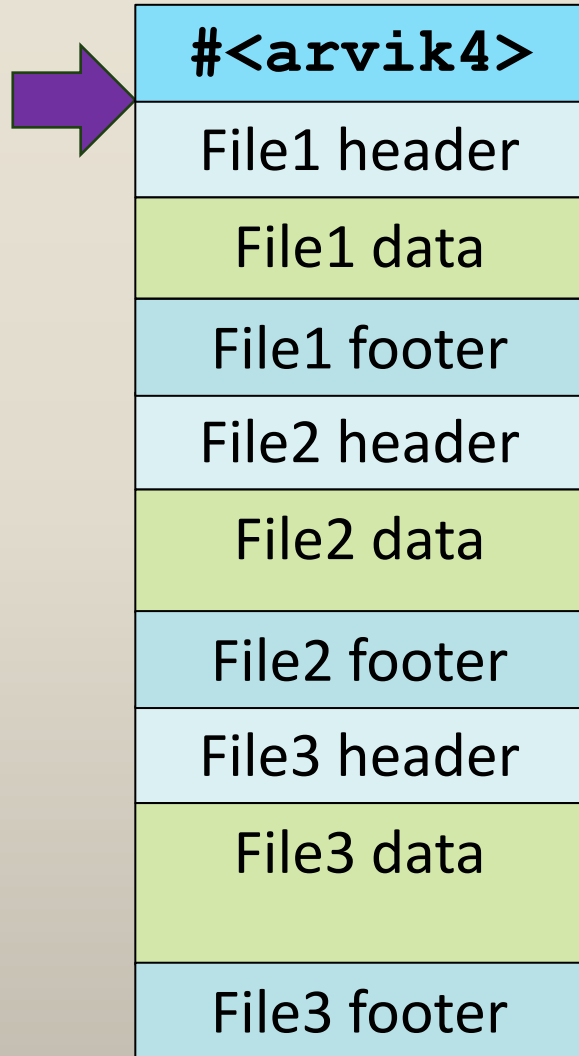


#<arvik4>
File1 header
File1 data
File1 footer
File2 header
File2 data
File2 footer
File3 header
File3 data
File3 footer

```
char *filename = NULL;
    // getopt() goodness goes in here
... ..
int iarch = STDIN_FILENO;

// if there is a -f filename in argv,
// you'll need to open() that file and
// assign the file descriptor to iarch.
if (filename != NULL ) {
    iarch = open(filename, O_RDONLY);
}
```

A Little Small TOC - 3



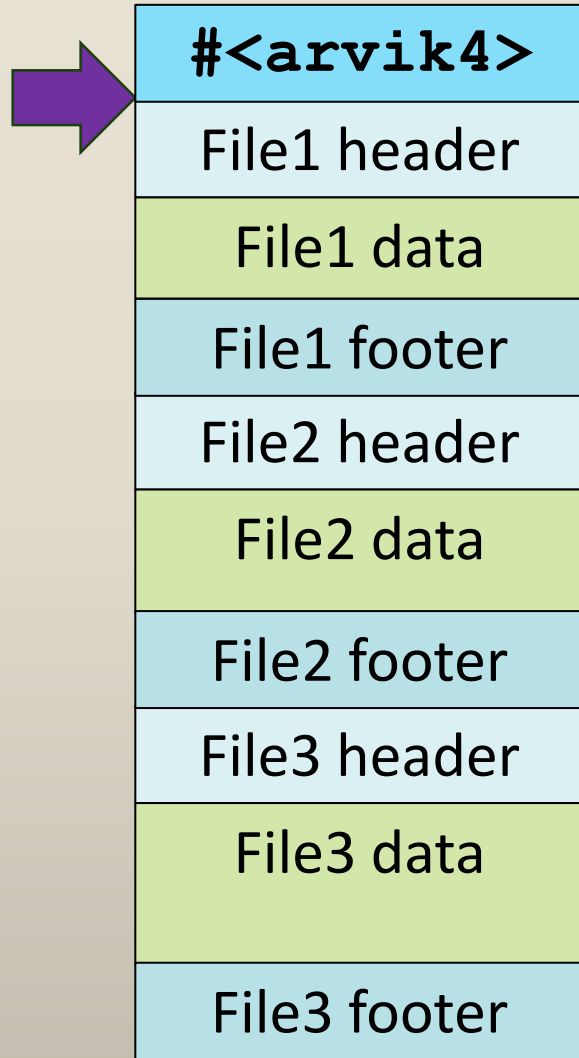
```
char buf[100] = {'\0'};
```

```
// validate tag
```

```
read(iarch, buf, strlen(ARVIK_TAG));
```

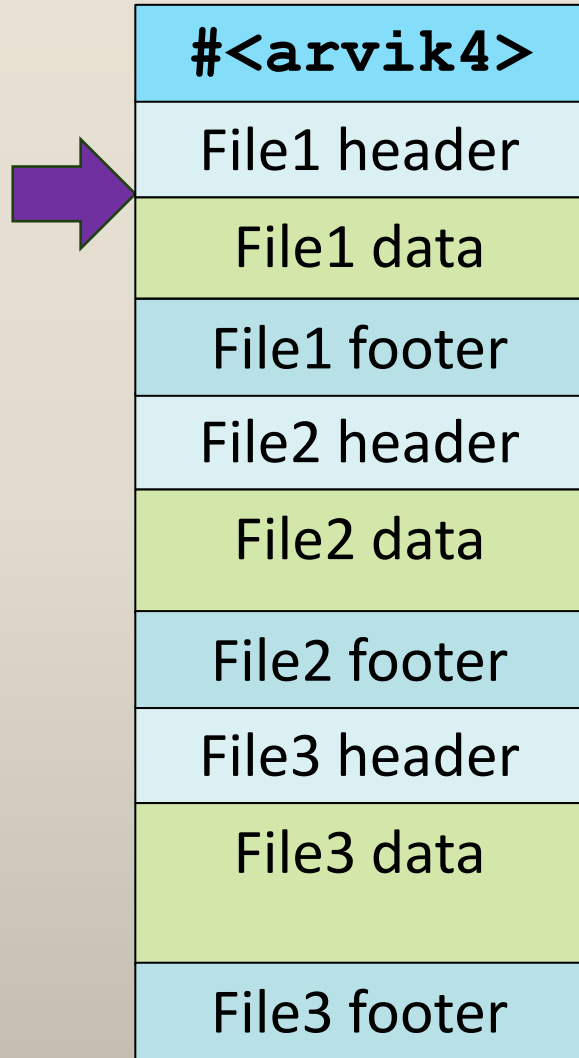
```
if(strcmp(buf, ARVIK_TAG, strlen(ARVIK_TAG)) != 0) {  
    // not a valid arvik file  
    // print snarky message and exit(1).  
    fprintf(stderr, "snarky message\n");  
    exit(EXIT_FAILURE);  
}
```

A Little Small TOC - 4



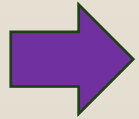
```
arvik_header_t md;  
char *back_pos = NULL;  
// process the archive file metadata  
while (read(iarch, &md, sizeof(arvik_header_t)) > 0 {  
    // print archive member name  
}
```

A Little Small TOC - 4



```
arvik_header_t md;  
char *back_pos = NULL;  
// process the archive file metadata  
  
while (read(iarch, &md, sizeof(arvik_header_t)) > 0 {  
    // print archive member name  
    memset(buf, 0, 100);  
    strncpy(buf, md.arvik_name, ARVIK_NAME_LEN);  
}
```

A Little Small TOC - 4



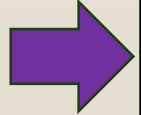
#<arvik4>
File1 header
File1 data
File1 footer
File2 header
File2 data
File2 footer
File3 header
File3 data
File3 footer

```
arvik_header_t md;  
char *back_pos = NULL;  
// process the archive file metadata
```

```
while (read(iarch, &md, sizeof(arvik_header_t)) > 0 {  
    // print archive member name  
    memset(buf, 0, 100);  
    strncpy(buf, md.arvik_name, ARVIK_NAME_LEN);  
    if ((back_pos = strchr(buf, ARVIK_NAME_TERM)) ) {  
        *back_pos = '\\0';  
    }  
}
```

```
}
```

A Little Small TOC - 5



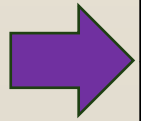
#<arvik4>
File1 header
File1 data
File1 footer
File2 header
File2 data
File2 footer
File3 header
File3 data
File3 footer

```
arvik_header_t md;  
char *back_pos = NULL;  
// process the archive file metadata
```

```
while (read(iarch, &md, sizeof(arvik_header_t )) > 0 {  
    // print archive member name  
    memset(buf, 0, 100);  
    strncpy(buf, md.arvik_name, ARVIK_NAME_LEN);  
    if ((back_pos = strchr(buf, ARVIK_NAME_TERM)) ) {  
        *back_pos = '\0';  
    }  
    printf("%s\n", buf);  
}
```

```
}
```

A Little Small TOC - 6



#<arvik4>
File1 header
File1 data
File1 footer
File2 header
File2 data
File2 footer
File3 header
File3 data
File3 footer

```

arvik_header_t md;
arvik_footer_t mf;
char *back_pos = NULL;
// process the archive file metadata
while (read(iarch, &md, sizeof(arvik_header_t)) > 0 {
    // print archive member name
    memset(buf, 0, 100);
    strncpy(buf, md.ar_name, ARVIK_NAME_LEN);
    if ((back_pos = strchr(buf, ARVIK_NAME_TERM))) {
        *back_pos = '\0';
    }
    printf("%s\n", buf);
    lseek(iarch, atoi(md.arvik_size)
        + (atoi(md.ar_size) % 2 == 0 ? 0 : 1)
        , SEEK_CUR);
}

```

A Little Small TOC - 6

#<arvik4>

File1 header

File1 data

File1 footer

File2 header

File2 data

File2 footer

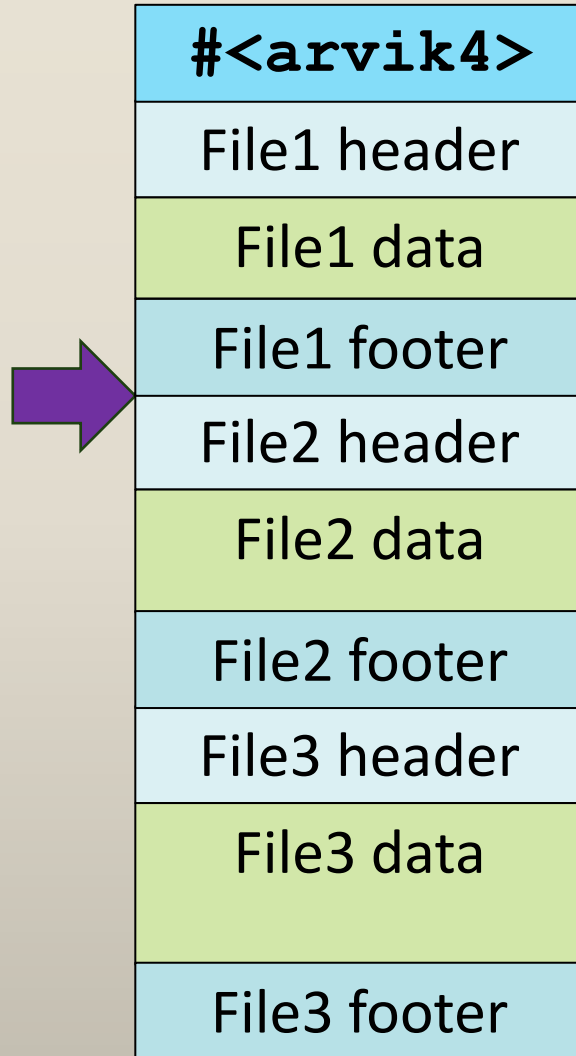
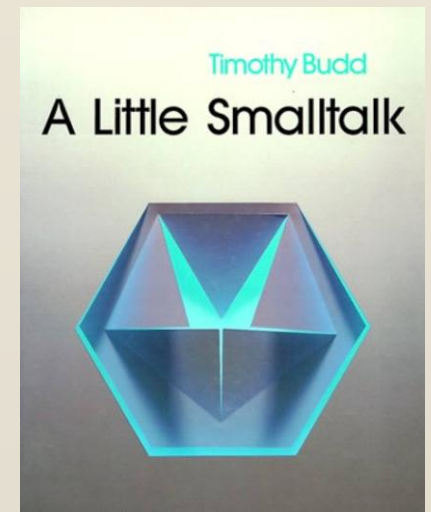
File3 header

File3 data

File3 footer

```
arvik_header_t md;
arvik_footer_t mf;
char *back_pos = NULL;
// process the archive file metadata
while (read(iarch, &md, sizeof(arvik_header_t)) > 0 {
    // print archive member name
    memset(buf, 0, 100);
    strncpy(buf, md.ar_name, ARVIK_NAME_LEN);
    if ((back_pos = strchr(buf, ARVIK_NAME_TERM))) {
        *back_pos = '\\0';
    }
    printf("%s\\n", buf);
    lseek(iarch, atoi(md.ar_size)
        + (atoi(md.ar_size) % 2 == 0 ? 0 : 1)
        , SEEK_CUR);
    read(iarch, &mf, sizeof(arvik_footer_t));
}
```

A Little Small TOC - 7



```
// we finished processing all the archive  
// members in the archive file.  
// the call to read() returned 0, indicating that  
// we hit end-of-file
```

```
if (filename != NULL ) {  
    close(iarch);  
}
```

Functions I used

```
close()  
exit()  
fchmod()  
fprintf()  
fstat()  
futimens()  
getopt()  
localtime()  
lseek()  
memcmp()  
memcpy()  
memset()
```

```
open()  
perror()  
read()  
sprintf()  
strchr()  
strftime()  
strlen()  
strncpy()  
strtol()  
umask()  
write()
```

You need to create and manage a couple instances of the `arvik_header_t` datatype (described in the `arvik.h` file). You also have a footer (`arvik_footer_t`) to manage.

A couple system/library functions have special types as well:

- **struct passwd**: used with the `getpwuid()` function
- **struct group**: used with the `getgrgid()` function
- **struct stat**: used with the `fstat()` function
- **struct timespec**: used with `localtime()` and `futimens()`
- **struct tm**: used with `localtime()` and `strftime()`