Lab 03: C Programming, mycp, mycat and mywc

Preliminaries

In this lab you will write a sequence of small C programs, culminating in writing a word count program like wc, called mywc. **You are likely to complete Task 1 - Task 3 in the lab period**, and will probably be able to start Task 4 and Task 5. You will definitely need to complete the mywc program, which should also be completed **individually**, outside of lab.

Lab Learning Goals

In this lab, you will learn the following topics and practice C programming skills.

1. Functions and Call by Value
2. File streams: Opening and Closing files
3. Command Line Arguments
4. Implementing our own word count mywc

Lab Setup

Run the following command

```
~aviv/bin/ic221-up
```

Change into the lab directory

```
cd ~/ic221/labs/03
```

All the material you need to complete the lab can be found in the lab directory. All material you will submit, you should place within the lab directory. **Throughout this lab, we refer to the lab directory**, which you should interpret as the above path.

Submission Folder

For this lab, all scripts for submission should be placed in the following folder:

```
~/ic221/labs/03/
```

This directory contains two sub-directories; examples and src. In the examples directory you will find any source code in this lab document. All lab work should be done in the src directory.

- **Only source files found in the folder will be graded**.
- **Do not change the names of any source files**

Finally, in the top level of the lab directory, you will find a README file. You must complete the README file, and include any additional details that might be needed to complete this lab.

Compiling your programs with clang and make

While gcc is the standard C compiler, it's output and error reporting is somewhat stale and old fashion. Instead, we will use the clang compiler, which is exactly like gcc, except it is better for educational purposes. To demonstrate the difference, consider the following code with a clear error;

syntaxerror.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
  int bob = 10;

  printf("%d\n", bab); //<-- oops

}
```

We can compare the two outputs from the compilers, which both report the error in understandable terms. However, clang is a lot more programmer friendly, and will make your C program experiences a lot easier.

```
#> gcc syntaxerror.c -o syntaxerror
syntaxerror.c: In function ‘main’:
syntaxerror.c:7:18: error: ‘bab’ undeclared (first use in this function)
syntaxerror.c:7:18: note: each undeclared identifier is reported only once for each function it appears in
```

```
#> clang syntaxerror.c -o syntaxerror
syntaxerror.c:7:18: error: use of undeclared identifier 'bab'; did you mean 'bob'?
  printf("%d\n", bab);
                 ^~~
                 bob
syntaxerror.c:5:7: note: 'bob' declared here
  int bob = 10;
      ^
1 error generated.
```

For this lab, we have provided you with a Makefile to help you compile your programs. To compile an individual program, such as your mywc program, type:

```
#> make mywc
```

To compile everything, just type make. Later in the course, you will be required to develop your own Makefiles. Before submission, you should clean your src directory by typing:

```
#> make clean
```

C Functions

The definition of C functions are mostly the same as C++ functions. A function is simply a code block that takes input, describe as arguments, processes that information, and returns some output. For example, here's a simple hello world.

hellofunc.c

```
#include <stdio.h>

void hello_function(){
  printf("Hello World");
}

int main(int argc, char *argv[]){
  hello_function();
}
```

Here the hello_function() does not take any input nor return any output, which is fine. In the case when a function does not take input or output, it is declared with a void keyword. However, you'll often find it more useful for your function to return a value. The type of that value must be declared in the function definition.

addone.c

```
#include <stdio.h>

int addone(int a){
   return a+1;
}

int main(int argc, char *argv[]){
   printf("%d\n", addone(10)); //prints 11
}
```

The addone() function takes an integer value, assigned to the variable a, and returns the increment of the input value, an int. A function can take as input any type, include pointers and structure data types, but the declaration of the function must be explicit.

Function Declaration

The C programming language is strongly-typed; that means, it checks the types of functions and its input/output during compilation. The compiler will report an error if it doesn't know the type of the function (the type of its input values and return values) or if the types of the input and output of the functions do not match. For example, consider the simple program:

typeerror.c

```
#include <stdio.h>

int main(int argc, char *argv[]){
  int a = 10;
  printf("%f\n",a); //expecting float, got int
}
```

The typing constructs also carry over to functions. You also have to declare your functions ahead of time so that the compiler knows what type to expect. Below is a common mistake:

funcdeclare_wrong.c

```
#include <stdio.h>

int addtwo(int a){
   return addone(addone(a)); //addone() hasn't been declared yet
}

int addone(int a){
   return a+1;
}

int main(int argc, char *argv[]){
   printf("%d\n", addtwo(10)); //prints 12
}
```

In the program below, the function addtwo() is declared before addone(), and, thus, at this part of the program the compiler is unaware of the type of addone() wich will result in the compiler warning.

```
funcdeclare_wrong.c:4:10: warning: implicit declaration of function 'addone' is invalid in C99 [-Wimplicit-function-declarati
  return addone(addone(a)); //addone() hasn't been declared yet
         ^
1 warning generated.
```

Instead, you have to declare the function prototype ahead of time, so that the compiler knows what types the function takes.

funcdeclare.c

```
#include <stdio.h>

int addone(int); //declaring the prototype of addone()
int addtwo(int); //declaring the prototype of addtwo()

int addtwo(int a){
   return addone(addone(a)); //addone() is now known here
}

int addone(int a){
   return a+1;
}

int main(int argc, char *argv[]){
   printf("%d\n", addtwo(10)); //prints 12
}
```

In general, it is good practice to declare all your function prototypes at the top of your program. In larger programs, function prototypes are declared in header files, which are included via the `#include` compiler directive.

Pass-by-Value Semantics

One major difference between C and C++ is that C does not have a pass-by-reference mechanism. That is, in C++ you can declare a function like `void addone(int &a)` which takes a reference. In C, all functions are always pass-by-value, which means that the values of the variables are **copied** to the functions.

Let's see how pass-by-value affects code execution in the example:

passbyvalue.c

```
#include <stdio.h>

int addone(int);

int addone(int a){
   a = a + 1; //different a!
   return a;
}

int main(int argc, char * argv[]){
  int a = 10;
  addone(a);
  printf("%d\n",a); //prints 10!
}
```

In `addone()`, the argument `a` is not the same `a` as was passed to it in `main()`. Pass-by-value dictates that the *value* of the variable `a` is copied, and that happens to be assigned to a variable named `a` in the `addone()` function.

How do you, then, affect the value of a variable from a function? You use pointers by passing a *pointer value* which is the address of the variable

passbyvalue_pointer.c

```
#include <stdio.h>

void addone(int *);

void addone(int *p){ //takes a pointer to a
  *p = *p + 1;
  return;
}

int main(int argc, char * argv[]){
  int a = 10;
  addone(&a); //pass the address of a
  printf("%d\n",a); //prints 11!
}
```

As a stack diagram, we can model the above program like this:

```
         addone(&a)         *p = *p + 1           return

(main)       |  (main)           |  (main)           |  (main)
.---.----.   |  .---.----.       |  .---.----.       |  .---.----.
| a | 10 |   |  | a | 10 | <-.   |  | a | 11 | <-.   |  | a | 11 |
'---'----'   |  '---'----'   |   |  '---'----'   |   |  '---'----'
             | ------------- |   | ------------- |   |
             |  (addone)     |   |  (addone)     |   |
             |  .---.---.    |   |  .---.---.    |   |
             |  | p | .-+----'   |  | p | .-+----'   |
             |  '---'---'        |  '---'---'        |
```

Before the call to addone(), the value of a is 10. The call to addone() passes the address of a, &a as the argument, which results in the pointer p referencing a. The assigning to the derefernced pointer also changes the value of a, incrementing it 11. Upon the return of addone(), the change to value of a is persistent because the increment happend at the address of a.

Task 1

Work in the file src/swap.c.

1. Complete the function swap() that takes pointers to two int's and swaps their values via pointer derferencing. Test your function in main.
2. Create a function swap_pair() which takes a pointer to the struct pair and swaps the order of the pair using your swap() function.

When correct, the output of the program should be as follows:

```
#>./swap
BEFORE SWAP: a: 10 b: 20
AFTER SWAP: a: 20 b: 10
----------------------
BEFORE SWAP: p.first: 50 p.second: 60
AFTER SWAP: p.first: 60 p.second: 50
```

(**HELPFUL HINTS**) Recall that when you have a pointer to a structure, to access it's element you use the -> operator. For example, a print_pair function would look like this:

```
print_pair(pair * p){
  printf("first: %d second: %d\n", p->first, p->second);
}
```

However, if you are working with the data type itself, then you use the . operator to access items, like in:

```
pair p;
p.first = 50;
p.second = 60;
printf("first: %d second: %d\n", p.first, p.second);
```

C Arrays and Strings

Arrays

An **array** is a contiguos region of memory for a sequence of data types. We declare arrays statically using the [ ] symbols and a size, and you can also reference and assign to an array using the [ ] symbol.

```
int array[10]; // array of integers of size 10
int i; //declared outside for loop
for(i=0;i < 10; i++){
 array[i] = i*10;
}
```

Arrays and pointers are closely linked, and, in fact, an array variable is a special type of pointer whose value cannot change. When you declare an array:

```
int array[10];
```

The value of the variable array is the memory address of the start of the contiguous memory region of 10 integers.

```
               .---.
array --> |   |  array[0] == *(array+0)
               +---+
               |   |  array[1] == *(array+1)
               +---+
               |   |  array[2] == *(array+2)
               +---+
               :   :  etc.
               '   '
```

When you index into an array, you are effectively you are following the pointer plus the index. That is, the value of `array[i]` is to following the pointer to the address of the start of the array, and then move `i` steps further, and then dereference that address to get the value. The concept of pairing arrays and pointers in this style is called **pointer arithmetic** and is an incredibly powerful tool of C programming.

Strings

An array of char's that is NULL terminated is a **string**.

```
char str[10] = "Hello!"
```

The char array, or string, `str` can store up to 10 chars, and thus strings of length 9. The last byte is used to NULL terminate the string; '\0' is the NULL character. In memory the string or buffer would like this:

```
          0   1   2   3   4   5    6   7   8   9    indexes
        .---.---.---.---.---.---.----.---.---.---.
str --> | H | e | l | l | o | ! | \0 | ? | ? | ? |
        `---'---'---'---'---'---'----'---'---'---'
                     char array
```

The '?' indicate that those memory places are unknown, since they haven't been written to yet. This why arrays are called buffers because they have a buffer of space for the data. If a buffer is overflowed, then all sorts of problems exist. Consider this simple example and output.

```
#include <stdio.h>

int main(){
  char str[10];
  fscanf(stdin,"%s",str);
  fprintf(stdout,"I read \"%s\"\n",str);
  return 0;
}
```

```
$> gcc io4.c
$> ./a.out
Hello
I read "Hello"
$> ./a.out
Hello???
I read "Hello???"
$> ./a.out
TheRainInSpainFallsMainlyOnThePlain
I read "TheRainInSpainFallsMainlyOnThePlain"
Segmentation fault
```

When the input exceed the buffer, which was only of size 10, then the additional characters of the input string "TheRainInSpainFalssMainlyOnThePlain" must be written somewhere. It turns out that it can overwrite other segments of code, causing a memory violation and the dreaded segfault. Worse, these kinds of bugs lead to security vulnerabilities and exploits which can make your program run unauthorized code. Simply, overflowing your buffer is bad, very bad.

Declaring Strings

When we declare strings, C programmers tend to be a bit inconsistent. For example, all the following are declaring strings of some form:

```
char s1[] = "I'm a string";
char s2[100] = "I'm also a string";
char * s3; //also called a string
```

The reason for this is that a string is just an array of `char`'s. All the above can reference an array. The declaration of s1 and s2 are both arrays, one with a fixed size and one based on the size of the assigning string. Also s3 is called a string even though it's not explicitely an array, and this is because of the tight relationship between pointers and arrays. A pointer can reference the start of an array, and generally when we use the `char *` decleration, we are referencing a string not just a pointer to a single `char`.

When we declare functions that take strings, we tend to use the `char *` decleration rather than the array declaration. For example:

```
void func( char * str);
```

And when you read manual pages where a string is requested, it may also be written like this:

```
size_t  strlen(const char *s); //from: man strlen
```

The `const` means that it wants a string, but the function won't alter the initial start of the string.

Iterating over Strings with Pointers

There are two standard ways for iterating over strings. The first is treating a string like an array. For example:

```
char str[] = "Hello World!";
int i;

for(i = 0; i < strlen(str) ; i++){
  printf("%c", str[i]);
}
```

Here, we declare an iterator index `i`, and use a standard for loop to iterate through the array. The string library function `strlen()` will provide the length of the string, the number characters prior to the `NULL` termination.

An alternative and common method for string iteration is using pointer arithmetic. Recall that in C, anything other 0 is true. `NULL` is actually the same thing as zero, and since `NULL` is false and all strings terminate with `NULL`, you can iterate over strings using pointer arithmetic. For example here is the same program with pointer arithmetic.

```
char str[100] = "Hell World!"; // the string is shorter than array
char *p;
for(p = str; *p; p++){ //when p points to NULL,
                       //i.e., the end of the string, the condition is false
                       // and the loop exits
  printf("%c", *p);
}
printf("\n"); // add a new line
```

The pointer `p` is initially pointing to the start of the string array, and it is incremented by one address for each loop, `p++`. Once it reaches the end, it is point to `NULL`, which we can check for with a dereference. I know it looks a bit strange, but it actually works!

While using pointer arithmetic is really convenient, it can get you into trouble. Consider what happens when the string is longer than the buffer? The loop will just keep going because it never reached the `NULL` character.

Task 2

Work in the file src/stirng$_{ex}$.c.

1. Complete the function `mystrcpy()` which takes as input two `char *`, `to` and `from`, and copies the string at `from` to the string at `to`.
2. Complete the function `reverse()` which takes a `char *` as input and reverses the order of the string.

For both of these functions you may use array access to perform string operations. To assist, you may use the string library function `strlen()` which will give the length of a string, not including the terminating NULL.

For **5 point Extra Credit**, complete your `mystrcpy()` and `reverse()` methods using only pointer arithmetic. Indicate this in your `README` file.

File I/O

We will use two different kinds of file I/O protocols in this course: file descriptors and file streams. The file descriptor protocol is slightly more complicated, and we'll cover that later in the course. In this lab, we'll use the simpler and higher level file stream protocol.

A file stream is declared as a `FILE` pointer:

```
FILE * file; //file pointer
```

You then open files using the `fopen()` function and close a file using the `fclose()` function. For example, below will read a file formated with numbers:

```
FILE * file;
int a,b,c;

if((file = fopen("path/to/file", "r")) == NULL){
  fprintf(stderr,"ERROR: fopen: file doesn't exist?\n");
  exit(1); //exit the program with error condition 1
}

while(fscanf(file, "%d %d %d", &a,&b,&c) == 1){
  //do something with a b and c
}

fclose(file); //close the file
```

The mode of a file, e.g. "r", indicates how the file can be used. The "r" mode from above is for "read", which means the file is opened for reading and the file stream is placed at the beginning of the file. The write mode "w", means to truncate the file (delete the current contents) and set the file for writing, starting at the beginning of the file. The write mode will also create the file if it doesn't already exits. Here are some other relevant modes:

- "r" : read mode
- "w" : write mode, truncates file and writes from beginning
- "a" : append mode, open file for writing without truncating, and beginning writing at the end of the file.
- "w+" : Open a file for reading and writing, and the file is truncated if it exists.
- "r+" : Open a file for reading and writing, the file is **not** truncated if it exists and writing occurs at the start of the file.

When working with files, many different errors can occur, such as the file not existing when opening for reading or not having permission to open a file or write to a file. You should **always** check the error conditions when opening a file. Upon error, `fopen()` returns `NULL`, which is easy to compare against in an `if` statement. It is customary to check for errors in your program, report the error, and exit with an error condition.

In addition to reading files with `scanf()` you can also use `fgetc()` and `fgets()`, but be sure to read the manuals for those functions. There exists a special value to denote the end of file, or EOF. You can compare against EOF to see if you've read the end of the file when using `fgetc()` or you can check the return value of `fscanf()`.

```
int c;
wile( (c = fgetc(file)) != EOF){
  // do something with c
}
```

Task 3

You will complete two small programs, `save.c` and `recall.c` found in the `src` directory of the lab folder.

1. The program `save` will read from `stdin` and save what is read to a file called `saved.out`. It should report an error if it can't open `saved.out` for writing.
2. The program will try and open the `saved.out` file and write it's contents to the screen. It will report an error if `saved.out` does not exist.

Here is some sample usage to compare against:

```
#> cat hemmingway.txt | ./save
#> ls saved.out
sved.out
#> diff hemmingway.txt saved.out
#> diff hemmingway.txt saved.out #will show nothing if the same
#> ./recall
One hot evening in Padua they carried him up onto the roof and he could look out over the top of the town. There were ch
(...)
#> rm saved.out
#> ./recall
ERROR: Cannot open output file for reading
```

Command Line arguments

Ok, you might have been wondering what the deal was with the argc and argv input to the main() function. This is how command line arguments are passed to the program. The argc is the count of the arguments, and the argv is an array of strings, one for each argument.

```
//              .--- The number of command line arguments
//              v
 int main(int argc, char *argv[])) { return 0;}
//                         ^
//  Array of strings of    |
//  command line        ---'
//  arguments
```

```
           .----.
argv -->  | .--+--> name of executable
          |----|
          | .--+--> arg 1
          |----|
          | .--+--> arg 2
          |----|
          .    .
          :    :
          |----|
          | .--+--> argc
          |----|
          | \0 |  NULL last value
          '----'
```

Just like in bash scripting, the first argument argv[0] is always the name of the executable, and the argv[1] is the first argument. At this point, it becomes trivial write a program that can enumerate the command line arguments.

```
int main(int argc, char * argv[]){
  int i;

  for(i=0; i<argc; i++){
     printf("arg %d: %s", i, argv[i]);
  }

  return 0;
}
```

Task 4

1. Write your own version of the cp function, mycp which takes two command line arguments, from and to, and copies a file byte-by-byte using fgetc() and fputc(). Your verision of mycp should report an error on the following two conditions:
   a. The from file cannot be opened with read permission
   b. The to file cannot be opened with write permission
   c. An invalid number of arguments was provided.

   Here is some sample usage of mycp to compare against:

```
#> ./mycp BeatArmy.txt CopyArmy.txt
#> cat BeatArmy.txt
Go Navy!
#> cat CopyArmy.txt
Go Navy!
#> ./mycp BADFILE copy
ERROR: Cannot open input file for reading
#> ./mycp copy BADFILE
ERROR: Cannot open output file for reading
#> ./mycp copy
ERROR: Invalid number of arguments
#> ./mycp
ERROR: Invalid number of arguments
```

2. Write your own version of cat function, mycat which has the same operations as cat.

    a. Without arguments, it should read from `stdin` and write to `stdout`

    b. With file arguments, it should attempt to open those files and write them to stdout in the order provided

    c. If an argument is - then it should read from `stdin` for that argument. To do this comparison of the argument to "-" use `strcmp()`, look it up in the manual for details.

    Here is some sample usage of `mycat` to compare against:

```
#> cat BeatArmy.txt | ./mycat
Go Navy!
#> ./mycat BeatArmy.txt
Go Navy!
#> ./mycat BeatArmy.txt GoNavy.txt
Go Navy!
Beat Army!
#> ./mycat BeatArmy.txt - < GoNavy.txt
Go Navy!
Beat Army!
#> ./mycat BeatArmy.txt - BeatArmy.txt < GoNavy.txt
Go Navy!
Beat Army!
Go Navy!
#> ./mycat BeatArmy.txt BADFILE GoNavy.txt
Go Navy!
mycat : Cannot open file 'BADFILE' for reading
Beat Army!
#> ./mycat BeatArmy.txt BADFILE GoNavy.txt 2> /dev/null
Go Navy!
Beat Army!
```

Word Count: mywc

In this part of the lab, you will complete the `mywc` program which functions much like the `wc` program, with some simple modifications, namely, that it can take input from files and `stdin` at the same time using the `cat` style syntax. You will use all the small parts of the above tasks to complete this one, and we expect it to be done outside of class.

Here is the usage for mywc:

```
#> ./mywc -h
mywc [OPTIONS] [file] [...]

Print the number lines, words, and bytes of each file or from standard input
All count information is written to standard out when no options are provided, and
by default data is read from standard input if no file is provided. To read from
standard input and a file, indicate standard input using '-'

OPTIONS:
         -h      print USAGE
         -l      print line count
```

```
          -w      print word count
          -c      print byte count
```

Parsing Aguments

The implementation of `mywc` is not about parsing arguments, so we have provided you with two helper functions for argument parsing.

```
/**
 * parse_args(int argc, char * argv[], struct mywc_opt_t * opts);
 * returns: index of remaining argv[]  values
 *
 * Set the opts structure with apropriate flags and return
 **/
void parse_args(int argc, char * argv[], struct mywc_opt_t * opts){ \\... }

/**
 * print_opts(int argc, char * argv[], struct mywc_opt_t * opts)
 *
 * Debug function to print the settings for mywc
 **/
void print_opts(int argc, char * argv[], struct mywc_opt_t * opts){ \\... }
```

The `pars_args()` function will parse the arguments and fill the `mywc_opt_t` structure with the settings. To help in understanding the flags, we have provided the `print_opts()` which print the options.

mywc data types.

There are two data types defined as structures that your program will use.

```
/**
 * Data type for mywc options
 **/
struct mywc_opt_t{
  int all;
  int lines; //== 1 if counting lines
  int words; //== 1 if counting words
  int bytes; //== 1 if countying bytes
  int f_index; //index into argv for start of file arguments
};

/**
 * Data type for mywc results
 **/
struct mywc_res_t{
  int lines; //number of lines
  int words; //number of words
  int bytes; //number of bytes
};
```

You will use these structures to store results and the options. Most of this should be fairly straight forward except for the `f_index` option. The `f_index` is the index into the `argv` array where the filenames start following the options, like `-l`. See the `print_opts()` function for an example of how to use this.

Counting lines, words, and bytes

This is the hard part of the lab, and although it does not require a lot of code, it will require some thought to do so efficiently. Here are some useful hints:

- You will need to check if a character is a newline, to do so you can compare against the newline '\n' to count lines.
- Each character is 1 byte, so just counting the number of characters is the same as determining the size of the file in bytes.
- To determine words, you will need to be able to identify white space, anything that separates two words. You can use the `isspace()` function for that. You will also probably want to use a variable to determine if the previous read character was whitespace to better determine word boundaries.

Task 5

Implement the word count program `mywc` in the file `mywc.c`. In addition to complete the program logic in `main()`, you are require to fill in the following functions:

1. `count()` : Given a a file pointer and a pointer to a `mywc_res_t` structure, it will read the file and fill in the results in the structure, counting lines, words, and bytes.
2. `print_res()` : Given options setting in `mywc_opts_t`, results in `mywc_res_t`, and the name of file read as a `char *`, print the results to `stdout`.

Here is some sample output to compare against.

```
#> ./mywc GoNavy.txt
    1       2       11      GoNavy.txt
#> ./mywc hemmingway.txt
    13      633     3235    hemmingway.txt
#> ./mywc hemmingway.txt GoNavy.txt
    13      633     3235    hemmingway.txt
    1       2       11      GoNavy.txt
    14      635     3246    TOTAL
#> ./mywc hemmingway.txt - GoNavy.txt < dickens.txt
    13      633     3235    hemmingway.txt
    19202   161009  936251  -
    1       2       11      GoNavy.txt
    19216   161644  939497  TOTAL
#> ./mywc < dickens.txt
    19202   161009  936251
#> ./mywc hemmingway.txt BADFILE
    13      633     3235    hemmingway.txt
mywc: File not found: BADFILE
#> ./mywc hemmingway.txt BADFILE 2> /dev/null
    13      633     3235    hemmingway.txt
#> ./mywc -l hemmingway.txt - GoNavy.txt < dickens.txt
    13      hemmingway.txt
    19202   -
    1       GoNavy.txt
    19216   TOTAL
#> ./mywc -l -w hemmingway.txt - GoNavy.txt < dickens.txt
    13      633     hemmingway.txt
    19202   161009  -
    1       2       GoNavy.txt
    19216   161644  TOTAL
#> ./mywc -c hemmingway.txt - GoNavy.txt < dickens.txt
    3235    hemmingway.txt
    936251  -
    11      GoNavy.txt
    939497  TOTAL
#> ./mywc -w hemmingway.txt - GoNavy.txt < dickens.txt
    633     hemmingway.txt
    161009  -
    2       GoNavy.txt
    161644  TOTAL
```

(EXTRA) C programming style

1. All variable declarations occur at the top of function
2. Use meaningful variable and function names that express functionality, lower case preferred with underscore separators
3. Comment generously, but not too much. Good code can speak for itself.
4. Use white space to separate different logical parts of your program to increase readability
5. Al [illegible] eck your error conditions, and report those errors to stderr.