CSC 230 Project 4

Reading List Manager

For this project, you get to write a program that will help you mange a reading list of books (say, to help select books to read over the summer). It will maintains a catalog of available books read in at program start-up. By entering commands for the program, the user can view the entire catalog of books, or just the books at a particular reading level. The user can choose books to add to their reading list and later remove them if they change their mind.

The sample execution below shows how you can run the program. The bold text is input typed by the user. Here, we're telling it to read a list of books from the input file, `list-d.txt`. We ask it to list the entire catalog (10 books in this case), then to display just the books with a reading level between grade level 9.0 and 10.5. After that, we add a few books to our reading list and display the reading list. Finally, we remove one book from the list, take another look at the list, and enter the quit command to terminate the program.

```
$ ./reading list-d.txt
cmd> catalog
catalog
   ID                                  Title               Author Level   Words
   42 The Strange Case of Dr. Jekyll and M.. Stevenson, Robert ..  11.0   25740
  120                        Treasure Island Stevenson, Robert ..   9.2   68267
  160 The Awakening, and Selected Short St..          Chopin, Kate   8.8   63991
  161                  Sense and Sensibility          Austen, Jane  12.3  118578
  219                      Heart of Darkness       Conrad, Joseph   9.0   37902
  768                      Wuthering Heights        Bronte, Emily   9.9  115874
  829 Gulliver's Travels into Several Remo..      Swift, Jonathan  19.3  104293
 1184 The Count of Monte Cristo, Illustrated     Dumas, Alexandre  10.8  459021
 2814                              Dubliners         Joyce, James   8.0   67546
 3207                              Leviathan       Hobbes, Thomas  20.6  213304

cmd> level 9.0 10.5
level 9.0 10.5
   ID                                  Title               Author Level   Words
  219                      Heart of Darkness       Conrad, Joseph   9.0   37902
  120                        Treasure Island Stevenson, Robert ..   9.2   68267
  768                      Wuthering Heights        Bronte, Emily   9.9  115874

cmd> add 120
add 120

cmd> add 1184
```

```
add 1184

cmd> add 829
add 829

cmd> list
list
   ID                                 Title               Author Level   Words
  120                        Treasure Island Stevenson, Robert ..   9.2   68267
 1184 The Count of Monte Cristo, Illustrated     Dumas, Alexandre  10.8  459021
  829 Gulliver's Travels into Several Remo..      Swift, Jonathan  19.3  104293
                                                                   13.1  631581

cmd> remove 1184
remove 1184

cmd> list
list
   ID                                 Title               Author Level   Words
  120                        Treasure Island Stevenson, Robert ..   9.2   68267
  829 Gulliver's Travels into Several Remo..      Swift, Jonathan  19.3  104293
                                                                   14.2  172560

cmd> quit
quit
```

As with recent projects, you'll be developing this one using git for revision control. You should be able to just unpack the starter into the p4 directory of your cloned repo to get started. See the Getting Started section for instructions.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

For this project, I used data based on Project Gutenberg. Thanks to the CORGIS project project for making this data available.

Rules for Project 4

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. For this assignment, we're putting some constraints on the functions you'll need to define, the data structures you'll use and how you're going to organize your code into components. Still, you will have lots of opportunities to design parts of your solution and to create additional functions to simplify your implementation.

Requirements

This section says what your program is supposed to be able to do, and what it should do when something goes wrong.

Program Execution

The `reading` program expects one or more filenames on the command line. Each of these files should contain a list of books that the program can read into its catalog at startup. If the program is run with invalid command-line arguments (e.g., no filenames given on the command line), it should print the following usage message to standard error and exit with a status of 1.

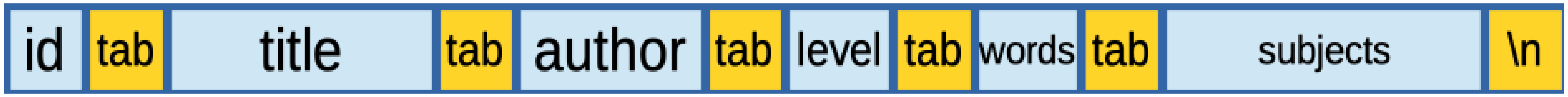
```
usage: reading <book-list>*
```

If the program can't open one of the given files for reading, it should print the following message to standard error and exit with a status of 1. Here, *filename* is the name of the file given on the command line. The program should report the first filename on the command line that it can't successfully open (i.e., if there are multiple filenames on the command line that can't be opened, it just needs to report this error for the first one that can't be opened).

```
Can't open file: filename
```

Book List Format

At program start-up, the `reading` program reads in a *catalog* of books. On the command line, it is given one or more filenames for files containing *book lists*, stored in a particular format. Each line of a book list file describes one book. A book description consists of six fields, with tab characters (ASCII 0x09) separating the fields. The first field is an integer ID unique to the given book. The next string is a title for the book (a string), then an author for the book (a string). The next field is a real number inditing the difficulty of the book, given as a grade level (e.g., 9.5 would be a grade level between the 9th grade and 10th grade). The next field is the number of words in the book and the last field is a string listing various keywords for what the book is about.



Format of a line of a Book List

Some books have an empty subject field. That's OK, but all the other fields should be non-empty. Your program will only use the subject field if you're doing the extra credit part of the assignment. Otherwise, your program can just skip over this field as it reads in a book list.

Some of the title and author fields are fairly long, but you will only need to store the first 38 characters of the title and the first 20 characters of the author name. This is described in the Book Listing section below.

The program should process the book list files in the same order they are given on the command line. Within each book list, it should process books in order from the first line to the last line of the file. The order for processing these files matters for error reporting. If there is something wrong with a book list, the program should report the first error it encounters.

A book list file can contain any number of book descriptions, one per line. If the format of the book list is invalid (e.g., if a line is missing one of the expected fields or if one of the numeric fields can't be parsed), then it should print the following message to standard error and exit with a status of 1. Here, *filename* is the name of the file containing the bad book description.

```
Invalid book list: filename
```

Every book should have a unique numeric ID. If the program encounters more than one book with the same ID (even if other fields like the title or author are different), it should print the following message to standard error and exit with a status of 1. Here, ID is the book ID that occurred more than once. The program should detect duplicate IDs, whether they occur within the same book list file or across two different book lists.

```
Duplicate book id: ID
```

Reading List

As the user interacts with the `reading` program, they can select books from the catalog to add to their *reading list*, a subset of books the user plans to read. The catalog is the set of all books available, and the reading list is the subset of the catalog that the user has selected.

Book List Output

A few user commands are intended to list books, either from the catalog or from the reading list. The output format for these reports is mostly the same. It consists of a header describing each of the five fields (like the example shown below). After the header, the report lists one book per line. Each book is reported as a book ID in a 5-character field, a book title in a 38 character field, a book author in a 20 character field, a book reading level in a 5 character field, with one fractional digit of precision and finally a word count given in a 7 character field. Each of these fields has a single space separating them.

```
  ID                                  Title               Author Level   Words
  42 The Strange Case of Dr. Jekyll and M.. Stevenson, Robert ..  11.0   25740
 120                        Treasure Island Stevenson, Robert ..   9.2   68267
 160 The Awakening, and Selected Short St..         Chopin, Kate   8.8   63991
 161                    Sense and Sensibility         Austen, Jane  12.3  118578
 219                      Heart of Darkness       Conrad, Joseph   9.0   37902
 768                      Wuthering Heights        Bronte, Emily   9.9  115874
 829 Gulliver's Travels into Several Remo..      Swift, Jonathan  19.3  104293
1184 The Count of Monte Cristo, Illustrated    Dumas, Alexandre  10.8  459021
2814                              Dubliners         Joyce, James   8.0   67546
3207                              Leviathan       Hobbes, Thomas  20.6  213304
```

For book titles or authors that are too long to fit in their field width, you will print as much of the title as you can, and then print two periods instead of the last two characters of the field, to indicate that the whole title was too long to fit. You can see this in the “Gulliver’s Travels” title and in the author “Robert Louis Stevenson” author above. Here, we print just the first 36 characters of the “Gulliver’s Travels” title, then print two periods at the end, making the overall length exactly 38 characters. We’ll do the same thing with the author name, but using 20 characters as the field width.

User Commands

After start-up, the `reading` program reads commands typed in by the user. Each command is given as single line of user input. For each command, the program will prompt the user with the following prompt. There’s a space after the greater-than sign, but you probably can’t see it in this web page.

```
cmd> 
```

After the user enters a command, the program will echo that command back to the user on the next output line. This is mostly to help with debugging your programs. If we’re capturing program output to a file, then things typed by the user don’t go to the output file (user inputs show up on the terminal, but they’re not part of the program’s output). By echoing each command, our output files will include a copy of each command the user typed, making it easier to see what the program was asked to do. So, for example, if the user typed in a command like the following, the program would echo a copy of the command on the next line:

```
cmd> level 9.0 10.5
level 9.0 10.5
```

The user can type any of 6 (or 7) available commands: `catalog`, `level`, `add`, `remove`, `list` and `quit`. There is also a `subject` command that can be implemented for extra credit. These commands are described below. Each valid command starts with one of the keywords listed above. For some commands, the keyword must be followed by one or more parameters. There may be one or more whitespace characters at the start of the command, between the keyword and the parameters, between parameters or at the end of the command.

If the user enters an invalid command, the program should print the following message to standard output (not standard error), ignore the command and prompt the user for another command. Invalid commands would be those that start with something other than the 6 (or 7 for extra credit) keywords listed above, or if the commands parameters weren't correct.

```
Invalid command
```

After the first prompt, the program should print a blank line before prompting the user for another command. This is shown in the sample execution at the start of this project description. It's just to provide a little separation between the output for consecutive commands.

The program should terminate when it is given the `quit` command or when it reaches the end-of-file on standard input. In the case of the quit command, the program should echo the command back to the user (like all the other commands). In the case of end-of-file, there's no command to echo, so the program should just terminate.

Catalog command

If the user enters the catalog command, the program should print out all the books in the entire catalog in the format described in the "Book List Output" section above. Books should be sorted by their ID field, least to greatest.

If there are no books in the catalog (this could happen if the program was given an empty book list file to read), the program should print the following message to standard output and then prompt for another command.

```
No matching books
```

Level command

The level command requires two real-number parameters, a low value and a high value. It lists books from the catalog with a reading level at least as high as the low value and no higher than the high value. Output should be given in the format described in the "Book List Output" section above, ordered by reading level, from low to high. Books with the same reading level should be sorted by ID.

For example, the user could enter the following `level` command, with the following response from the program. Notice that the two books with a reading level of 8.8 are ordered by ID number.

```
cmd> level 8.5 10.0
level 8.5 10.0
```

```
  ID                                  Title               Author Level   Words
4300                                Ulysses         Joyce, James   8.6  264835
  74           The Adventures of Tom Sawyer          Twain, Mark   8.8   70796
 345                                Dracula         Stoker, Bram   8.8  160693
 158                                   Emma         Austen, Jane   9.6  157439
1400                     Great Expectations     Dickens, Charles   9.9  184398
```

The level command would be invalid if it was missing a parameter, or one of it's parameters couldn't be parsed as a double value, or if its first parameter was greater than its second parameter. If the range of reading levels doesn't contain any books, the program should print a line to standard output saying "No matching books", like in the following example:

```
cmd> level 3.0 4.0
level 3.0 4.0
No matching books
```

Subject command

The subject command is for **extra credit**. For this command, the program will need to store the list of subjects for each book (the last field on each line in a book list). The user can enter the subject keyword, followed by a single word (a sequence of non-whitespace characters). The program will find all books in the catalog that contain that word as a substring in their subject field. It will print out just these books from the catalog in the format described in the "Book List Output" section above, with books that match the given word listed in order of ID. A book's subject field matches the given word even if the word is just a substring of a longer word in the book's subject field. For example, if the user entered "subject just", then it could match a book that had the word "just" in its subject field, or one that had the word "adjustment" in its subject field.

If there are no matching books in the catalog, the subject command should print the "No matching books" message to standard output, just like the catalog and level commands.

A subject command would be invalid if it didn't have a word after the `subject` keyword, or if it had more than one word after it.

Add command

The `add` command is for adding books from the catalog to the reading list. Books added to the reading list are added at the end, and adding a book to the reading list doesn't remove it from the catalog; it just puts that book on the reading list. The `add` command expects a book ID as a parameter. So, for example, the following command would add the book with an ID of 42 to the reading list.

```
add 42
```

An add command would be invalid if there wasn't an integer after the add keyword. If the integer doesn't match a book ID from the catalog, the program should print the following to standard output (where ID is the ID the user asked to add).

```
Book ID is not in the catalog
```

If the user gives the ID of a book that's already on the reading list, the program should print the following message to standard output (where ID is the ID the user asked to add):

```
Book ID is already on the reading list
```

Remove command

The `remove` command is for removing books from the reading list. As a parameter, it expects the ID of the book to be removed. It removes that book from the reading list, and the remaining books stay in the same order.

If the remove command isn't given a valid integer as a parameter, then it is an invalid command. If the parameter doesn't match ID of a book on the reading list, then it should print the following message to standard output, where ID is the ID of the book the user asked to remove.

```
Book ID is not on the reading list
```

List command

The list command shows the books on the reading list in the same format described in the "Book List Format" section above. Output for the list command should include one extra line at the end. This extra line should give the average reading level for all the books on the list, rounded to one fractional digit of precision and printed right under the column giving the reading level for each book. It should also report the total length (in words) for all the books on the list. The total length should be shown below the words column at the right end of the output.

For example, running the list command might look like the following.

```
cmd> list
list
   ID                                  Title               Author Level   Words
  120                        Treasure Island Stevenson, Robert ..   9.2   68267
 1184 The Count of Monte Cristo, Illustrated     Dumas, Alexandre  10.8  459021
  829 Gulliver's Travels into Several Remo..      Swift, Jonathan  19.3  104293
                                                                   13.1  631581
```

If the reading list is empty, the program should print a line to standard output saying “List is empty”. In this case, it won’t print the line with the average reading level and the total word length. So, for example, you might get the following response from a `list` command:

```
cmd> list
list
List is empty
```

Quit command and termination

The quit command doesn’t take any parameters. It should terminate the program. It’s entered like the following:

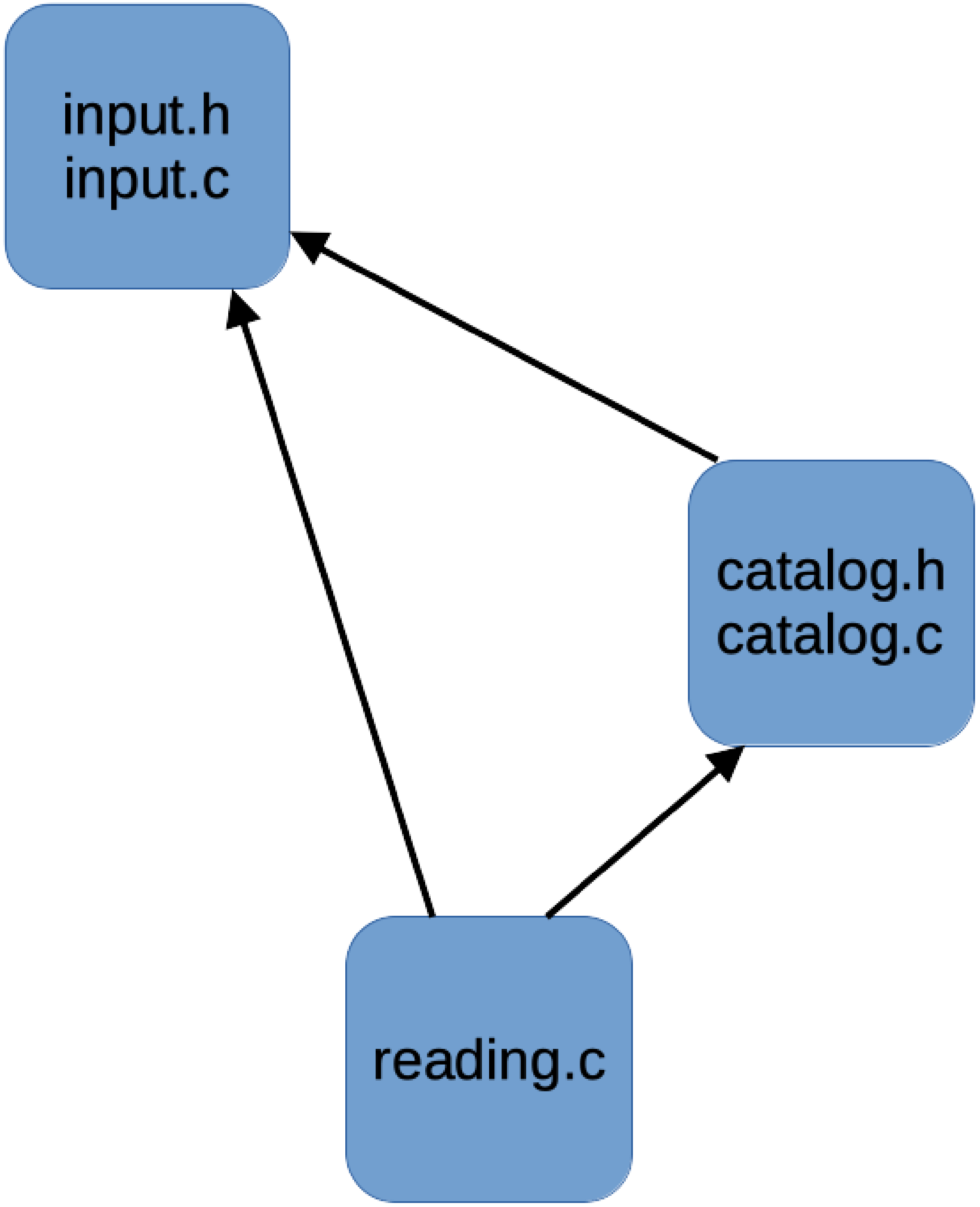
```
cmd> quit
```

The program should also terminate successfully if it reaches the end-of-file on standard input while it’s trying to read the next command.

Design

Program Organization

Your implementation will be organized into three components. The `input` component will help with reading input from the book list files and from the user. The `catalog` component will contain code for implementing books and the catalog. The `reading` component will contain main, code to read in user commands and the implementation for the reading list.

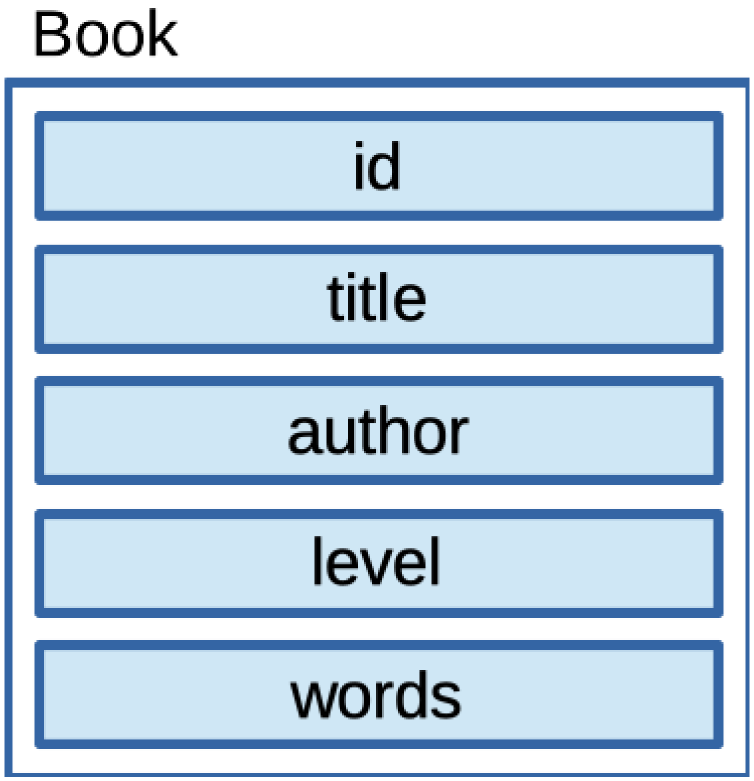


Components and Dependency Structure

The `input` and `catalog` components will each have a header file so other components can use types and functions defined by these components. The figure above shows the dependency structure of the project. The `catalog` component can use code provided by `input` and the main `reading` component can use code provided by both `input` and `catalog`.

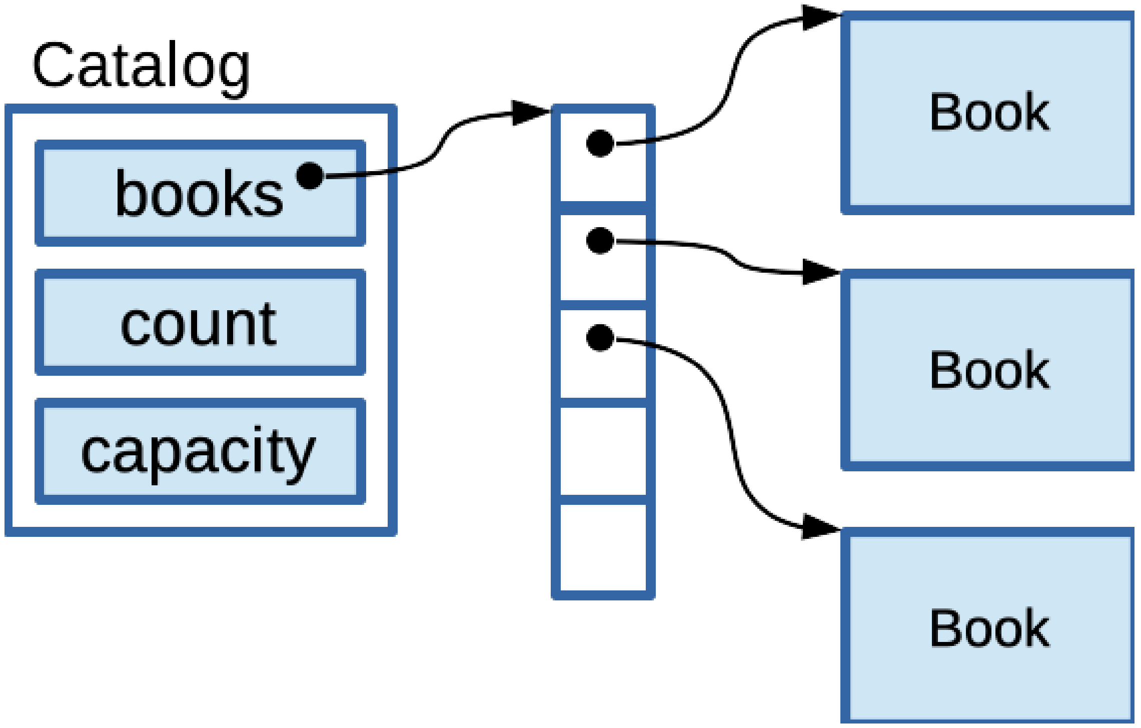
Book and Catalog Representation

This project is a good chance to get some experience using structs, dynamic memory allocation and resizable arrays. Each book will be represented by a struct with a field for each of the five values associated with a book. The title and author fields will be strings, the ID and word count can be stored as ints, and the reading level can be stored as a double. The title field just needs to be able to store a string of up to 38 characters. Although lots of titles are longer than this, the output of the program never reports more than 38 characters for a title, so you won’t need to store more than the first 38 characters. Likewise, the author can be stored in a char array with room for a string of up to 20 characters.



Book Represntation

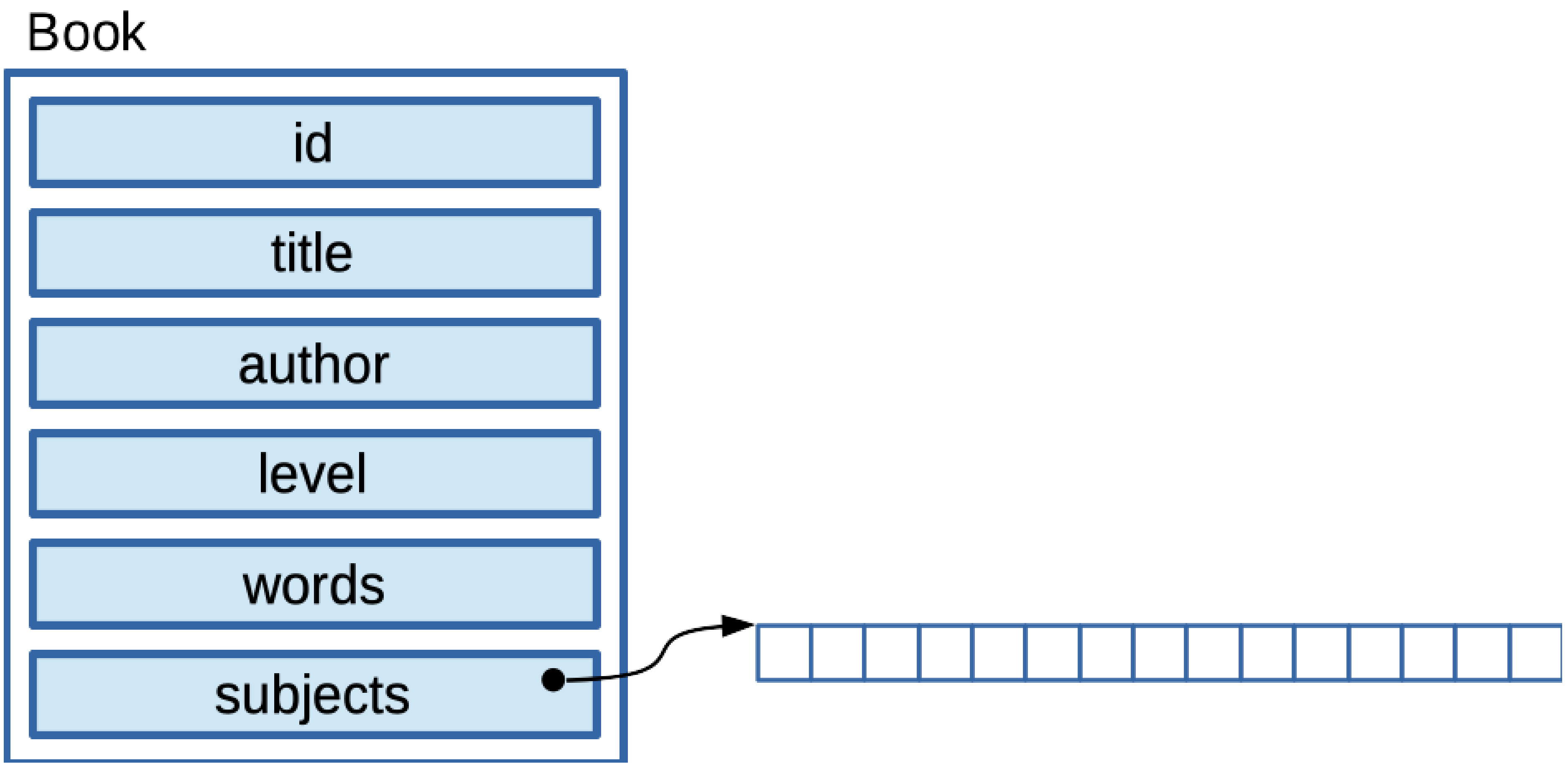
The catalog will be represented by its own struct, containing fields to store a resizable array of pointers to books. Each book will be stored in a block of dynamically allocated memory. Inside the catalog struct, you will use a resizable array of pointers to books to keep up with all the books on the catalog. The count and capacity fields are for maintaining the resizable array, for keeping up with how many books are in the catalog and for detecting when you run out of capacity and need to grow the array. Your resizable array should start with an initial capacity of 5, and it should double in capacity whenever the array needs to be enlarged.



Catalog Representation

Extra Credit Design and Implementation

If you do the extra credit part of this project, each book will need to store a string of subject keywords read from the reading list files. The subjects for a book may be a long string, so we’re not going to store it inside the book struct. Instead, the string will be stored in another block of dynamically allocated memory, and the book struct will just keep a pointer to this string. That way, the subject string can be exactly as long as it needs to be to hold whatever string is given in the book list input.



Book Represntation with Subjects

Note that some books have an empty subjects string. This is OK; it just means that those books will never show up in response to a subject query from the user.

For the extra credit, you may find the `strstr()` function useful for finding a short string inside a larger one. We will talk about this function briefly in class, but, if you want to use it for the extra credit part of the assignment, you may need to do some reading on your own. You'll find it on page 620 of your textbook, or, if you're on a Linux machine, you can just type `man strstr` at the shell prompt to look at the online documentation.

Reading List Representation

You will represent the reading list as a resizable array in the `reading` component. Like the catalog, this array should start with an initial capacity of 5 and it should double in size whenever it needs to grow.

You can represent your book list however you want to. If you want, you can store it inside a struct, like we're doing with the catalog, or you can just use some global variables inside the `reading` component to keep up with the reading list. If you do choose to use some global variables for the reading list, be sure to mark them as static. This will prevent possible name collisions with symbols defined elsewhere in your program.

Expected Functions

As part of your implementation, you will define and use the following functions. You can define more if you want to. Just try to put them in a component that's suitable for whatever they do and remember to mark them as static where you can (i.e., if they're not used outside the component where they're defined).

Your `input` component only needs to have one function.

- `char *readLine( FILE *fp )`
  This function reads a single line of input from the given file and returns it as a string inside a block of dynamically allocated memory. You can use this function to read commands from the user and to read book descriptions from a book list file. Inside the function, you should implement a resizable array to read in a line of text that could be arbitrarily large. If there's no more input to read, this function should return NULL.

Your `catalog` component should have the following 6 (or 7) functions.

- `Catalog *makeCatalog()`
  This function dynamically allocates storage for the catalog, initializes its fields (to store a resizable array) and returns a pointer to it.

- `void freeCatalog( Catalog *cat )`
  This function frees the memory used to store the catalog, including freeing space for all the books, freeing the resizable array of book pointers and freeing space for the catalog struct itself.

- `void readCatalog( Catalog *cat, char const *filename )`
  This function reads all the books from a book list file with the given name. It makes an instance of the Book struct for each one and stores a pointer to that book in the resizable array

- `void listAll( Catalog *cat )`
  This function lists all the books in the catalog, sorted by ID number. The `reading` component can call this in response to the user entering the `catalog` command.

- `void listLevel( Catalog *cat, double min, double max );`
  This function lists all the books with a reading level between the given min and max values (inclusive). Your `reading` component can call this when the user enters the `level` command. In the output, books should be sorted by reading level, and by ID if they have the same reading level.

- `void listSubject( Catalog *cat, char const *subject )`
  You only need this function if you're doing the extra credit. It reports all books where the given subject string occurs in he book's subjects field.

- `void listCatalog( Catalog *cat, bool (*test)( Book const *book, void const *data ), void const *data )`
  This is a static function in the catalog component. It is used by the listAll(), listLevel() and litSubject() functions to actually report the list of books in the right format. In addition to a pointer to the catalog, this function also takes a pointer to a function (test) and a pointer to an arbitrary block of data (data) to let the caller tell the function which particular books it should print out. This is described in more detail in the "Selecting Books to Report" section below.

Your `reading` component will contain main() and any other functions you need to parse command-line arguments and user commands.

Function Visibility

Any functions that are needed by a different component should be prototyped (and commented) in the header. Functions that don't need to be used by a different component should not be prototyped in the header, and should be marked static (given internal linkage), so they won't be visible to any other part of the program. This is like making the function an implementation detail of its component, something we could change if we wanted without affecting other parts of the program.

Sorting the Catalog

You'll use the standard library qsort() function to sort books, either by ID or by reading level (and ID). Using qsort() will make the sorting easier (and probably more efficient), but you have to help out qsort() by providing a pointer to a comparison function. We have some examples of this in the material from lecture 12, in the slides and in the sort.c example program.

To use qsort() you'll need to think about a few things. As usual, you'll need to write your own comparison function, one that takes two (const) void pointers, but knows that they're really pointers to two elements of the array inside the

Catalog struct. So, your function will need to cast these void pointers to pointers of the right type before it can start looking at the fields of the Book objects they point to. Remember that the comparison function gets pointers to two array elements (not copies of the values in two array elements, **pointers** to the elements). So, for example, since the array is full of pointers to Book structs, your comparison function will get two pointers to pointers to to Book instances. You have to define your comparison so it takes two void pointer parameters, but, internally, your comparison function will know that these are really pointers to pointers to Books. After casting the void pointers you get to these more specific types, you can access the fields of the Books in order to compare them.

You have to sort the catalog two different ways (for the level command vs for the catalog command), so you will need to implement two different comparison functions for sorting the books in the catalog.

Parsing User Commands

We're reading user input one line at a time. After we get a string containing a command, we will need to look inside this string to figure out what command the user typed. The sscanf() function will make it easy to do this. It works much like scanf() or fscanf(), but it parses input from a string instead of from a file. We'll cover this function in lecture 18, but you may want to look at the material for lecture 18 early (it should already be posted) so you can get started on the project earlier.

Selecting Books to Report

The `listCatalog()` function can be used to print any selected books from the catalog, so, it can be used by the three functions listAll(), void listLevel() and void listSubject() to print out the needed subset of the catalog. How does it know which books to report? Internally, it will call the provided test function for each Book in the catalog. If the test function returns true, listCatalog() prints that Book; otherwise, it doesn't. This lets client code use a single interface to print any subset of the Books. The client code just needs to provide a pointer to a function listCatalog() can use to decide what to print and what not to print. For example, to perform the `catalog` command, you can pass in a pointer to a test function that always returns true. To print Books in a range of reading levels, you can pass in a pointer to a function that checks the reading level and returns true if the Books reading level is in the range. To do this, we'll need to use the `data` parameter to listCatalog().

Notice that listCatalog() takes a void pointer data parameter, and the test function also takes a void pointer data parameter. This parameter is a mechanism for providing extra information the test function needs in order to do its job, like the range of reading levels needed for the `level` command When you call listCatalog(), you can pass in a pointer to anything you want as the data parameter (even NULL, if you don't need this parameter). The listCatalog() function will remember this parameter and will give this same pointer to the test function every time it calls it. This gives you a way to supply a pointer to anything your test function needs to answer the question, "Should we print this Book?". Inside each of your test functions, you will just need to convert the data value from a void pointer back to whatever it really points to before it can use it.

For example, to implement `level 5.5 8.5` command, our test function needs to know two values 5.5 and 8.5, so it can return true for books with a reading level in this range. That's what the data parameter is for. You can put 5.5 and 8.5 in a little struct, then pass the address of that struct as the last parameter to listCatalog(). You'll also give listCatalog() the address of a test function that knows how to check the reading level of a book against the range of values you provided via the data parameter. Each time listCatalog() calls your test function, it will pass it a copy of the data pointer you provided (i.e., a little struct in which you stored the range, 3.5 and 8.5). Inside your test, you can cast the data void pointer back to a pointer to your little struct, then use it to decide if the given Book should be printed. Or, if you don't want to define a little struct to hold the minimum and maximum reading level, you could use a little 2-element array, where you store the minimum and maximum reading level in elements of the array.

If you're doing the extra credit, you can use a similar technique to implement the `subject` command, except your test function will need a data value that tells it what the search string is (rather than a minimum and maximum value for the reading level range).

This is a common trick in C, using a void pointer to pass any information you need through general-purpose code and eventually back into code written for a specific purpose. When you take the operating systems class, you'll see a similar technique in the POSIX threads API, to pass arbitrary data to a new thread you're creating.

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your program, compiling each source file to an object file and then linking the objects together into an executable.

As usual, your Makefile should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what source files have changed. It should also have a `clean` rule, to let the user easily delete any temporary files or target files that can be rebuilt the next time make is run (e.g., the object files, the executable and any temporary output files).

In addition to the "-Wall" and "-std=c99" options that we normally include when we're compiling, be sure to include the "-g" flag. This will be useful when you try to use `gdb` or `valgrind` to help debug the program.

Testing

The starter includes a test script, along with test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program (using your Makefile) and see how it does against all the tests.

As you develop your program, you'll want to try it out with user input and see what it's doing on individual test cases. Until your Makefile is working, you should be able to execute the compiler directly with the following command (although this isn't as efficient as how your Makefile builds).

```
$ gcc -g -Wall -std=c99 reading.c catalog.c input.c -o reading
```

To try out your program on one of the provided test cases, you can run it as follows. Here, we're saving the program's standard output and standard error to two different files. All of the tests have an expected output file for standard output, and some of them (the ones that exit unsuccessfully) also have an expected output on standard error.

Here, we're running test 12 by hand. After running the program, we check its exit status to make sure it ran successfully (it should for this test). Then we check the output file to make sure we got what was expected. This particular test shouldn't print any output to the standard error stream.

```
./reading list-d.txt < input-12.txt > output.txt 2> stderr.txt
$ echo $?
0
$ diff output.txt expected-12.txt
$ cat stderr.txt
```

Memory Error and Leaks

Your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. Although it's not part of an automated test, we encourage you to try out your executable with valgrind. We certainly will when we're grading your work. Any leaked memory, use of uninitialized memory, access to memory outside the range of an allocated block or leaked files will cost you some points. Valgrind can help you find these errors before we do.

The compile instructions above include the `-g` flag when building your program. This will help valgrind give more useful reports of where it sees errors. To get valgrind to check for memory errors, including leaks, you can run your program like the following. This example runs the program on input 12, a test input that uses several of the supported commands. You can use similar commands to try your program on any input using valgrind.

```
$ valgrind --tool=memcheck --leak-check=full ./reading list-d.txt < input-12.txt
-valgrind output omitted-
```

If you want to run valgrind, you have to run it on each test case individually. You can't run the test script inside valgrind (well, you can, but then you will be getting valgrind output for the shell, rather than for the program you wrote). I've seen students try to run something like "valgrind test.sh". Just be aware that this won't work. You have to run valgrind like the example above.

Test Cases

The tests for your program use the following catalog files. These contain various-sized lists of books, along with some invalid book lists for testing error handling.

1. list-a.txt : This list just just contains one book.

2. list-b.txt : This list contains 5 books. This shouldn't require resizing the array of books maintained by your Catalog, and they're already sorted by ID number.

3. list-c.txt : This list contains 22 books, sorted by ID. With 22 books, this list should force the resizable array in the Catalog to resize twice while reading the list.

4. list-d.txt : This list contains 10 books that aren't sorted bi ID, so you will need to be able to sort them to report the catalog in the right order.

5. list-e.txt : This list contains 968 more books from among the most downloaded on Project Gutenberg.

6. list-f.txt : This list is like list-d.txt, except it has a book with an ID of 1080, a duplicate ID for a book on list-c.txt.

7. list-g.txt : This list is just like list-d.txt, except one of the lines is missing some fields.

We've prepared ?? tests for your program. Using the book lists above, they exercise the various commands your program is supposed to support, working from the easier ones to the more difficult tests and error cases. In developing your program, you may want to follow the order of these tests, adding the code to support the first test, then working toward the later (more complex) tests as you get the earlier ones working.

1. This test reads the list-a.txt file, but it runs the quit command immediately.

2. This test reads the list-b.txt file. It runs the catalog command then input ends at the end-of-file on the input, rather than a quit command (which should be OK).
3. This test reads the list-c.txt file, which should cause your catalog to have to resize its list of book pointers.
4. This test reads the list-d.txt file and runs the catalog command, which will require your program to sort books by ID.
5. This test reads 3 book lists given on the command line, list-a.txt, list-b.txt and list-c.txt.
6. This test reads book lists list-c.txt and list-d.txt, then uses the level command to list books with a reading level between 9.7 and 13.0.
7. This test reads book lists list-c.txt and list-d.txt, then uses the level command to list books with a reading level between 3.5 and 4.5. None of the books in these lists have a reading level in that range.
8. This test reads book lists list-c.txt and list-d.txt, then uses the level command to list books with a reading level between 5.0 and 27.0. All of the books in these lists have a reading level in that range.
9. This test reads book list list-d.txt. It adds a couple of books to the reading list, showing the list before and after.
10. This test reads book lists list-a.txt, list-b.txt, list-c.txt and list-d.txt. I adds all the books in all these lists to the reading list. That should require the array in the reading list to resize more than once.
11. This test is like the previous one, but, after adding all the books to the reading list, it removes 10 of them.
12. This test reads book list list-d.txt. It runs the same commands as the example shown at the start of the project description.
13. This is a large test. It loads more than 1000 books from input-a.txt .. input-e.txt. It runs the catalog and list command (with various parameters). Then it randomly adds and removes some books from the reading list, showing the list periodically.
14. This is a test for error handling. The command line arguments ask the program to open a book list file that doesn't exit.
15. This is a test for error handling. The program is run with invalid command-line arguments (no arguments).
16. This is a test for error handling. The standard input includes two invalid commands.

17. This is a test for error handling. It tries to add the same book to the reading list more than once, and it tries to remove a book from the reading list that isn't there.

18. This is a test for error handling. It reads list-c.txt and list-f.txt at startup, giving it two books with the same ID.

19. This is a test for error handling. It reads list-g.txt, which includes a bad book description.

Extra Credit Tests

If you do the extra credit part of the assignment, we're providing a couple of test cases you can use to check your program's behavior. The extra credit inputs are `input-ec-1.txt` ad `input-ec-2.txt`. For each of these, we're also providing an expected output file. You can try them using the following commands to see if your program is working for the extra credit part of the assignment.

```
$ ./reading list-d.txt < input-ec-1.txt > output.txt
$ echo $?
0
$ diff output.txt expected-ec-1.txt

$ ./reading list-d.txt < input-ec-2.txt > output.txt
$ echo $?
0
$ diff output.txt expected-ec-2.txt
```

Grading

The grade for your program will depend mostly on how well it functions. You also get to provide your own Makefile, and we'll expect your program to compile cleanly, to follow the style guide and to adhere to the expected design. Completing the extra credit can earn you some additional points.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **5 points**
- Behaves correctly on all tests: **80 points**
- Program follows the style guide: **20 points**
- Support for subjects and the subject command: **8 extra credit points**
- Deductions
  - Up to **-60 percent** for not following the required design.

- Up to **-30 percent** for failing to submit required files, submitting files with the wrong name or having extraneous files in the repo.
- Up to **-30 percent** for exhibiting file leaks, memory leaks or other memory errors.
- **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the `p4` directory of your repo. You'll submit by committing files to your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't already done this, go back to the assignment for project 2 and follow the instructions for for cloning your repo.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 4 starter. We're providing this file as a compressed tar archive, starter4.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p4/starter4.tgz
```

Temporarily, put your copy of the starter in the p4 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter4.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter to your repo. After you've successfully unpacked it, you may want to delete the starter from your p4 directory, or move it out of your repo.

```
$ rm starter4.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `reading.c` : source file for the top-level component, written by you.
- `catalog.c` : Implementation file for the catalog component, written by you.
- `catalog.h` : Header for the catalog component, written by you.
- `input.c` : Implementation file for the input component, written by you.
- `input.h` : Header for the implementation component, written by you.
- `Makefile` : the project's Makefile, written by you.
- `list-*.txt` : book lists for the tests, provided with the starter.
- `input-*.txt` : test inputs given to the program on standard input, provided with the starter.
- `expected-*.txt` : expected output from the program, provided with the starter.
- `estderr-*.txt` : expected standard error output for some test cases, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file provided with the starter, to tell git not to track temporary files specific to this project.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Of course, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to previous projects. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 4. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.

- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O.

- Use simple command-line tools to design, document, debug, and maintain their programs.

- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.

- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.

- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.

- Describe and demonstrate how to avoid the implications of common programming errors that lead to security vulnerabilities, such as buffer overflows and injection attacks.