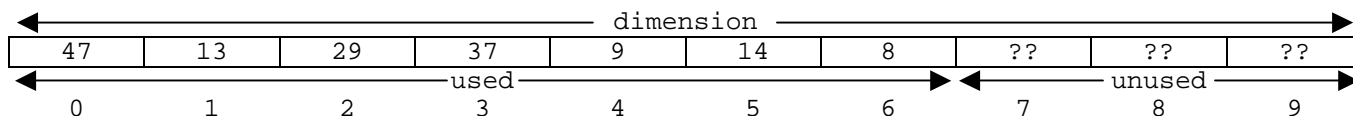


Managing a simple array:**Validating Array Indices**

Most interesting programs deal with considerable amounts of data, and must store much, or all, of that data on one time. The simplest effective means for doing that is to use an array. This project is about basic array management. In general, we think of an array as consisting of a range of index values, determined by the array dimension. Within that range, some index values correspond to data and some index values correspond to unused array locations. Something like this:



Here, we have an array of dimension 10, so the cells would be indexed 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Data is stored in only the first seven cells (indexed 0, 1, 2, 3, 4, 5, 6) while the last three cells (indexed 7, 8, 9) are unused.

When using an array in a program, it is usually necessary to pay careful attention to whether an index value lies within the used part of the array, within the unused part, or entirely outside of the array.

For this project, you will declare an array of integers, having dimension 25. The input file will specify how many cells of the array (beginning at index 0) are used. You will initialize the array so that each cell in the used portion stores a value read from an input file, and each cell in the unused portion stores the value 0. Your program will then process commands, described below, that require performing operations on the array.

Logically, some of those operations will be legitimate and some will not. Your program must determine whether a command is legitimate and carry it out or provide an error message. Commands may specify index values that are negative, or too large for the given dimension. Commands may specify index values that are physically valid, with respect to the array dimension, but that lie outside of the “usage” range. Any such index values are considered to be illegitimate. The description of the input file explains how you should indicate each type of illegitimate index.

The commands file:

The input file, named "Script.txt", will begin with a header line specifying the number of cells of the array that are considered to store valid data (the “usage”). The header line will begin with a label, which will be terminated by a colon (':') character as shown below.

The header line will be followed by one or more lines containing a whitespace-separated list of positive integer values to be used to initialize the used cells of the array. The number of values will exactly match the value given in the header line.

Each of the remaining lines will begin with a command word, followed immediately by a single tab character, and then by one or two whitespace-separated integer values, depending upon the particular command word. The commands file will be syntactically correct, but the commands may contain any of the logical errors described above.

Here is a description of each of the commands your program must recognize and process:

```
set <integer value> <target index>
    If the given index lies in the used part of the array, store the given value at that index and print a message
    confirming the operation. If the given index is unacceptable, print an error message indicating what is wrong.
```

```
show <source index>
    If the given index lies in the used part of the array, display the value stored at that index. If not, print an
    appropriate error message indicating what is wrong.
```

swap <source index> <target index>

If the source and target indices are both in the used part of the array, swap the values at the given source index and at the given target index. If the source index is unacceptable, print an error message about the source index. If the source index is acceptable but the target index is not, print an error message about the target index.

multiply <LHS index> <RHS index> <target index>

If the LHS and RHS indices and the target index are in the used part of the array, multiply the values at the given LHS and RHS indices and store the product at the given target index. If any of the indices are unacceptable, print an error message about the first one that has a problem.

exit

Stop processing the commands file immediately. This produces no output.

Note that the formatting and content of the output must match the sample output file given later in this specification. Here is a sample commands file:

```
Usage: 21
 143 24 13 9 98 51 23 71 50 82
 3 180 225 93 47 3 102 39 107 8
 41
show 0
set 17 0
show 0
set 17 -5
show 23
set 25 23
show 23
set 23 10
set 11 55
show 3
show 13
swap 3 13
show 3
show 13
show 0
multiply 2 10 0
show 0
multiply -1 17 4
multiply 14 55 4
multiply 14 4 55
swap 15 -4
swap 17 50
swap -4 17
swap 25 17
exit
```

The output file:

As usual, the output begins with a header identifying the programmer and the assignment. Following a blank line, print the array usage (as specified in the input file), labeled exactly as shown. After that comes the output, if any, produced for each command read from the commands file.

Note that you must use the exact notation shown above in your output. Do not use `setw()` when printing the array index values; just let them print with their natural length and no spaces between them and the surrounding brackets. Do not insert spaces except where they are shown above. Since each command, except `exit`, should produce a line of output, it is easy to match the output lines to the commands that inspired them.

The output file must be named "Trace.txt". Here is a sample output file generated from the sample input file given earlier:

```

Programmer: Bill McQuain
CS 1044 Project 6 Fall 2001

Used cells: 0 to 20
Array[0] == 143
Array[0] <-- 17
Array[0] == 17
Array[-5]: negative index
Array[23]: index unused
Array[23]: index unused
Array[23]: index unused
Array[10] <-- 23
Array[55]: index too large
Array[3] == 9
Array[13] == 93
Array[13] <--> Array[3]
Array[3] == 93
Array[13] == 9
Array[0] == 17
Array[0] == Array[2] * Array[10]
Array[0] == 299
Array[-1]: negative index
Array[55]: index too large
Array[55]: index too large
Array[-4]: negative index
Array[50]: index too large
Array[-4]: negative index
Array[25]: index too large

```

Handling script-driven computations:

There are a number of ways to parse the sort of commands file used in this project. We will describe one good approach (given what we've covered so far) here. Aside from any header lines, each line of the commands file has the form

```
<command string><tab>{<parameter>}*
```

where:

command string denotes any of the specified command words (or phrases)

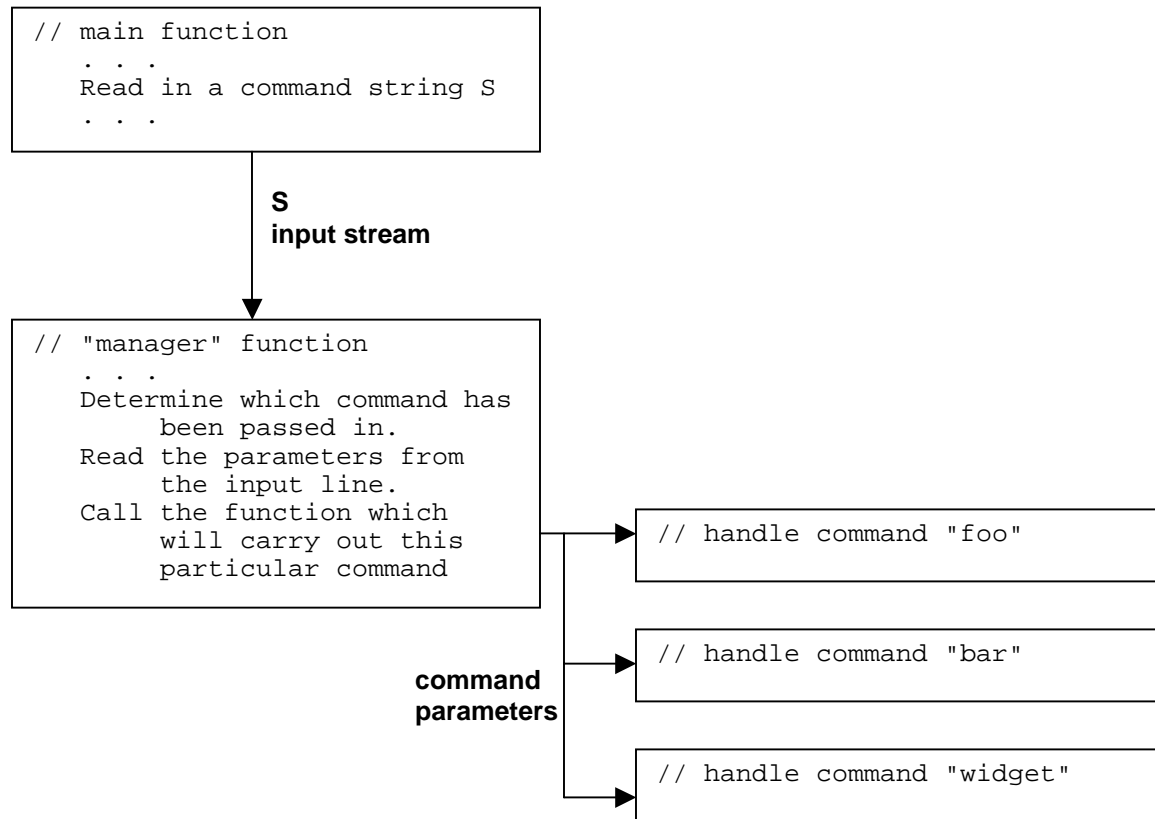
parameter denotes any valid parameter (index, integer value, etc.) for the given command string

The asterisk simply indicates that the term enclosed in the braces may occur zero or more times (actually, one or more times for most of the commands in this project).

Typically, the particular command string that is used will determine the number of parameters. In this project, for example, the `store` and `swap` commands each take two parameters, while the `show` command takes one and the `exit` command takes none. So, it is necessary to read the command string and determine what it is before the remainder of the line can be parsed.

Also, the logical significance of the parameters depends upon the particular command string that is used. For example, while `store` and `swap` both take two parameters, the first parameter to `store` is just a data value and the second is an index while both parameters to `swap` are index values. So, how we interpret the parameters after we've read them depends upon the particular command string that we just read.

Here is one good organization for processing this sort of command file. Each box represents a function, each arrow a function call. Some (but not all) of the parameters that need to be passed are indicated next to the call arrows.



Determining which command has been passed in requires comparing the string *S* to the known command strings. Once that has been done, we know how many parameters are expected (and their types), and we know what function to call to handle that command.

The design makes it relatively simple to add support for new commands, and logically isolates the code to read and handle each command in a sensible way.

Function requirements for this program:

You must make good use of functions in your implementation. There are many opportunities to do so in this project. For instance, you might use a function to initialize the array, a separate function to handle each command, a function to read in the a command string, a function to determine what the command is, a function to check the validity of an array index value, and functions to perform output.

Your design must include at least five user-defined functions, not counting `main()`. For reference, my solution uses ten user-defined functions. It is also important that you choose the appropriate parameter-passing mechanisms, especially when you pass the array to a function. Pass parameters by reference only when it is logically necessary; otherwise, pass parameters by value or by constant reference.

Evaluation:

Everything that was said in the specification for Project 1 about testing still applies here. Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. We may also be evaluating your submission of this program for documentation style and a few good coding practices. If so, this will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

Read the *Programming Standards* page on the CS 1044 website for general guidelines. You should comment your code in the same manner as the code given for the first two programming assignments. In particular:

- Write a header comment with your identification information, the required pledge statement (below), and a brief description of what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Adopt a consistent indentation style and stick to it.

Your implementation should also meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use named constants instead of literal constants when the constant has logical significance.
- Use C++ streams for input and output, not C-style constructs.
- Use C++ string variables to hold character data, not C-style character pointers or arrays.
- You must use a count-controlled loop to read the list initialization values, and a hybrid input-failure/sentinel controlled loop to manage the remainder of the input script.
- You must use an array in this program.
- You must not use global variables in this program. Global function declarations and global constants are OK.
- Design and implement functions, as specified above.
- Each function parameter should be passed using an appropriate protocol. Parameters should only be passed by reference if the called function needs to modify the actual parameter. Potentially large parameters, such as strings, should be passed by constant reference instead of by value (unless they need to be passed by reference).
- The first function implemented in your source file must be `main()` – so you must provide appropriate prototypes for each of your functions.
- You must include a header comment with the implementation of each function you write. The header comment should be formatted to match the sample function header posted on the *Programming Standards* page of the course website.

Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* can be found at: <http://ei.cs.vt.edu/~eags/Curator.html>

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
// On my honor:  
//  
// - I have not discussed the C++ language code in my program with  
//   anyone other than my instructor or the teaching assistants  
//   assigned to this course.  
//  
// - I have not used C++ language code obtained from another student,  
//   or any other unauthorized source, either modified or unmodified.  
//  
// - If any C++ language code or documentation used in my program  
//   was obtained from another source, such as a text book or course  
//   notes, that has been clearly noted with a proper citation in  
//   the comments of my program.  
//  
// - I have not designed this program in such a way as to defeat or  
//   interfere with the normal operation of the Curator System.  
//  
// <Student Name>
```

Failure to include this pledge in a submission is a violation of the Honor Code.