Handout #9, Project 1

Monday, October 24, 2005, 6:00PM.
Electronic submissions only.

In this project, you will develop a calculator for vectors.
Your program will read its input from standard input, and write its output to standard output. Your program should be called \texttt{vcalc}.

1. Conventions

When discussing program usage examples, \textbf{boldface text} represents program output, and \textit{italicized text} represents program input.

2. Program Versions

Two versions of the program are to be developed. In the first version, only two vectors can be stored at any given time by the user in the calculator’s memory, and these vectors are to be managed in your program as two one-dimensional arrays. In the second version of the program, up to 10 vectors can be stored at a time, and the vector storage is to be managed as a two dimensional array.

3. Overall Program Functionality

The calculator should operate by repeatedly prompting the user for a calculator command, and executing the command. The string that is to be displayed at the calculator’s command prompt is:

\texttt{VCALC(\textit{X})>}

where \textit{X} represents the number of the next command to be processed. For example, the initial prompt displayed by the calculator is \texttt{“VCALC(1)>”}. After reading and processing the first command, the prompt becomes \texttt{“VCALC(2)>”}, and so on. Note the space that follows the \texttt{“>”} character in the prompt.

4. Most Recent Result

The majority of \texttt{vcalc} commands are so-called “result-producing commands” (RPCs). An RPC operates on one or two operands and produces a single result, which is printed to standard output. The \texttt{vcalc} calculator keeps track internally of the result produced by the last (most recent) RPC. This most recent RPC result is referred to as the MRR (“most recent result”). Each time a new RPC is executed, the internal MRR is updated to hold the result produced by the new RPC.

5. Vector Identifiers

A user of \texttt{vcalc} refers to vectors symbolically using vector identifiers, which are lower-case alphabetic strings (strings consisting only of the lower-case alphabetic characters ‘a’, ’b’, …, ’z’). A vector identifier can contain at most 12 characters.
Vector values associated with vector identifiers are kept track of (stored in memory) by vcalc. The value associated with a vector identifier is initialized or updated using the st (set) command. A vector identifier is discarded from memory by the ut (un-set) command. When a vector identifier is discarded, memory is no longer reserved for it, and its associated storage space can be used by another vector identifier (through subsequent use of the st command). After "un-setting" a vector identifier using ut, the identifier is no longer defined (i.e., is no longer recognized) in vcalc.

At most VECTOR_MAX vector values can be stored in memory at any given time. That is, the number of vector identifiers that can be defined at a given time is VECTOR_MAX. In the first version of this project, VECTOR_MAX = 2, and in the second version VECTOR_MAX = 10.

The maximum vector length (maximum number of elements in a vector) is 10.

6. Precision
Vector elements should be maintained as double entities. The precision (number of digits after the decimal point) to which to print vector elements can be set by the user using the pn command. The default precision is 2, and the maximum allowable precision is 8.

7. Blank Lines
The calculator should ignore lines of input that consist only of white space (spaces, newlines, tabs, etc.). Such blank lines should not result in error messages or program termination.

8. Calculator Commands
Every vcalc command begins with a two-letter code that indicates the name of the command. Each command occupies a single line of input: a command cannot stretch across multiple input lines (i.e., across newline characters), and conversely, multiple commands cannot be specified on the same input line. Each command requires a fixed number of arguments to be specified along with the command. Depending on the command, zero, one, or two arguments must be specified. This section describes the set of vcalc commands. The set is grouped according to the number of arguments that must be specified along with the commands.

8.1 Zero-argument Command
There is only one command that does not require arguments:
• ex
  Exit. The ex command simply terminates the calculator program. No output is produced.

8.2 One-argument Commands
One-argument commands have the general form

<command> <argument>

where <command> is a symbolic placeholder for the command identifier (two-letter code), and <argument> is a symbolic placeholder for the command’s argument. For most of these commands, the argument specifies a vector.

• st
  Set. The argument of the st command is a vector identifier. This command sets the value associated with the given vector identifier to be the MRR. If the vector identifier is already defined, its current value is overwritten with the MRR. If the vector identifier is not defined, it is defined (reserved in memory), and initialized with the MRR. It is an error to execute the st command before any RPCs have been executed. It is also an error to execute the st command if the given vector identifier is not currently defined, and the number of vector identifiers that are currently defined is equal to VECTOR_MAX.
• **pn**

*Precision.* The precision command sets the precision (see Section 6) with which vector elements are printed with subsequent calls to the `pr` command. The argument must be an non-negative integer that is less than or equal to the maximum precision, as specified in Section 6.

• **pr**

*Print.* The print command simply displays a vector based on the present precision value (see Section 6). The command argument gives the vector identifier for the vector whose value is to be printed. The format for printing a vector is to print the vector on a single, separate line of output; to enclose the vector in parentheses, and to separate the elements by commas followed by single spaces. For example:

\[(3.22, -4.28, 9.00, 6.96)\]

is an example of output from the `pr` command, assuming a current precision specification of 2.

It is an error to try to print a vector that is not currently defined.

• **sr**

*Set MRR.* This command is used to set the value of the MRR to a user-specified vector value. The argument is a vector, which is specified as a comma-separated list of floating point numbers that is enclosed by parentheses. The command prints the specified vector \(X\) to the screen in the same format as the `pr` command, and sets the value of MRR to equal \(X\).

Example usage: \(sr (3.45, -3, 9.7, 10.5)\)

• **ut**

*Un-set.* The argument of the `ut` command is a vector identifier. See Section 5 for a specification of what this command does. It is an error if the given vector identifier is not currently defined.

• **br**

*Bar graph.* The `br` command takes a vector identifier as its argument, and prints a bar graph with one "bar" corresponding to each element \(x_i\) of the specified vector \(X = (x_1, x_2, \ldots, x_n)\). Dashes ('-' characters) are used to print the bars. The bars are printed on separate lines, starting with the bar for the first vector element \(x_1\), and then the bar for the second vector element \(x_2\), and so on. The number of dashes in a given bar is equal to the *floor* of the corresponding vector element, where the floor of a floating point number \(f\) is defined to be the largest integer that is less than or equal to \(f\).

It is an error for the specified vector to have any negative-valued elements (i.e., to have \(x_i < 0\) for any \(i\)).

For example, consider the following sequence of VCALC commands:

```
VCALC(1)> pn 2
VCALC(2)> sr (5, 7.2, 2.5)
VCALC(3)> st X
VCALC(4)> br X
```

This sequence of commands results in the following text being printed to standard output. The first line is produced as the output of the `sr` command, and the remaining lines are produced as a result of the `br` command.

\[(5.00, 7.20, 2.50)\]

-----

------

--

• One-argument computational commands: `ng`, `sm`, `rm`, `ab`.

Each of these commands takes a vector identifier as an argument, performs a unary vector operation, updates the MRR to equal the result of the operation, and prints the result to standard output in the same format as the `pr` command. It is an error if the specified vector is not currently defined.
If $Y = (y_1, y_2, \ldots, y_n)$ represents the vector specified by the command argument, then the computations performed by these commands are as follows.

— *ng* (negation): $\text{MRR} = (-y_1, -y_2, \ldots, -y_n)$.

— *sm* (sum): $\text{MRR} = \left(\sum_{i=1}^{n} y_i\right)$. Note that this result is always a one-element vector.

— *rm* (root-mean-square): $\text{MRR} = \left(\frac{\sum_{i=1}^{n} y_i^2}{n}\right)$. Note that this result is also always a one-element vector.

— *ab* (absolute value): $\text{MRR} = (|y_1|, |y_2|, \ldots, |y_n|)$.

### 8.3 Two-argument vector-vector computations

Most of the two-argument commands correspond to binary vector operations. Each of these operations takes two vector identifiers as arguments, performs a binary vector operation, updates the MRR to equal the result of the operation, and prints the result to standard output in the same format as the *pr* command. It is an error if either of the specified vectors is not currently defined. It is also an error if the vectors do not have the same length (same number of elements).

If $Y = (y_1, y_2, \ldots, y_n)$ and $Z = (z_1, z_2, \ldots, z_n)$ represent the vectors specified by the command arguments, then the computations performed by these commands are as follows.

— *ad* (addition): $\text{MRR} = ((y_1 + z_1), (y_2 + z_2), \ldots, (y_n + z_n))$.

— *sb* (subtraction): $\text{MRR} = ((y_1 - z_1), (y_2 - z_2), \ldots, (y_n - z_n))$. The second operand is subtracted from the first operand — element by element — to obtain the result.

— *mp* (multiplication): $\text{MRR} = ((y_1 z_1), (y_2 z_2), \ldots, (y_n z_n))$.

— *dv* (division): $\text{MRR} = \left(\frac{y_1}{z_1}, \frac{y_2}{z_2}, \ldots, \frac{y_n}{z_n}\right)$. It is an error for any of the elements of the second operand to be zero-valued (i.e., it is an error if $z_i = 0$ for one or more indices $i$).

— *ip* (inner product): $\text{MRR} = \left(\sum_{i=1}^{n} y_i z_i\right)$. Note that this result is always a one-element vector.

### 8.4 Scaling

The two-letter code *sc* represents the scaling command, which, like the commands in Section 8.3, takes two vector identifiers as its arguments. However, unlike the commands of Section 8.3, the first vector identifier must correspond to a single-element vector (i.e., a vector length of one) $Y = (y)$. The vector $Z = (z_1, z_2, \ldots, z_m)$ specified by the second argument can be of any length. This operation updates the MRR to equal the result of the operation, and prints the result to standard output in the same format as the *pr* command. It is an error for the vector corresponding to the first argument to have a length other than one.

The computation performed by this command is as follows.
Thus, the result is obtained by multiplying each element of the vector \( Z \) with the scalar value \( y \).

### 8.5 Summary of Commands

Table 1 provides a summary of `vcalc` commands. The first and second column give the two-letter code and a brief description, respectively, for each command. The third column specifies the number of command arguments. The fourth column indicates whether or not the command is an RPC.

#### 9. Error Handling

All of the errors specified in the description of the commands above are recoverable errors. This means that whenever it encounters such errors, `vcalc` reports a meaningful error message, stops processing the command (without any further output or any modification of the MRR value), and continues its processing with the next line of input.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Arguments</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>Exit</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>st</td>
<td>Set</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>pn</td>
<td>Precision</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>pr</td>
<td>Print</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>sr</td>
<td>Set MRR</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>ut</td>
<td>Unset</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>br</td>
<td>Bar graph</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>ng</td>
<td>Negation</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sm</td>
<td>Sum</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>rm</td>
<td>Root mean square</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>ab</td>
<td>Absolute value</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>ad</td>
<td>Addition</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>sb</td>
<td>Subtraction</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>mp</td>
<td>Multiplication</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>dv</td>
<td>Division</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>ip</td>
<td>Inner product</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>sc</td>
<td>Scaling</td>
<td>2</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Summary of commands.
All other errors (e.g., specifying a vector identifier that exceeds the maximum allowable length, giving a non-existent command specification, leaving out parentheses when specifying a vector value, and giving too many or two few arguments to a command) are fatal errors. When it encounters fatal errors, `vcalc` outputs a meaningful error message and exits.

10. Additional Requirements

Two versions of the program should be submitted (See Section 2). The version that allows only two vectors should be submitted as `vcalc-light.c`, and the full-featured program (up to ten vectors) should be submitted as `vcalc.c`.

Preprocessor constants should be used for all constant values that relate to program limits, such as the maximum number of vectors that can be kept in calculator memory at any given time, and the maximum number of characters in a vector identifier.

Adherence to course coding conventions is required throughout your implementation.

11. Grading

- Correctness under error-free input: 40%
- Handling of recoverable errors: 10%
- Documentation (comments): 10%
- Organization and clarity of code (apart from coding conventions): 10%
- Use of course coding conventions: 20%
- Handling of fatal errors: 10%