Cmpt 214
Term 1 (Fall), 2016/17

Assignment #2

Construction sign Warning: still subject to minor r
evision ...

Out: October 11, 2016

Originally Due: 23:55, October 25, 2016

Now Due: 23:55, November 1, 2016

Overview

In this assignment, you will work toward developin
g a program in "procedural C++" (C++ not including
 Classes/objects, templates, etc. -- see below) ca
lled vecalc which is a simple, interactive vector
calculator. The functionality of the program is de
scribed below. What students need to complete and
submit is described under "Assignment Requirements".

The focus of this assignment is not so much on con
structing a program that provides a noteworthy fun
ctionality, but rather on practising defensive pro
gramming, using good programming style, and on fol
lowing the test-driven development (TDD) process.
As such, simply having an assignment that works co
rrectly will be insufficient to get a good mark.

A "vector" in the context of this assignment has m
eaning taken from linear algebra: it is simply a o
ne-dimension array.

Procedural C++

Procedural C++ is a subset of C++ that does not in
clude classes/objects, templates, etc. Note that t
his rules out use of objects such as cin and cout,
 but booleans, syntatic elements of C++ statements

(such as variables declared within for statements
), new and delete operators (when used to create s
tructures or variables of built-in types such as i
nt or float), static variables and integer constan
ts are allowed. Please see the lecture notes from
the first day of class for greater detail, and fee
l free to ask a the lab TA or the instructor if yo
u are uncertain as to eligibility of a feature of C++.

Functional Requirements

The vecalc program operates on a retained vector o
f (single-precision) floating-point values. It alt
ernates between two states, an input state and a c
alculation (or calculator) state. The program begi
ns execution in the former state, wherein it creat
es an empty vector, prompts the user for a number
of elements in the vector, and builds up a vector
with elements obtained one-at-a-time from the user
. The program then switches to calculation state.
In calculation state vecalc accepts operations and
commands from the user and executes them. After m
ost commands the program remains in calculation st
ate. On a "clear" command ("c"), the program frees
up the memory used by the current vector (i.e. de
letes that exiting vector) and switches back to in
put state.

Elements of a vector are to be of type float. A ve
ctor can contain from 0 up to 65535 elements.

Input is taken from stdin and normal output goes to
stdout. Error output goes to stderr.

The operations and commands supported by vecalc in
calculator state are as follows:
q : quit; same functionality as "e" (end) below

c : clear; free up the dynamically-allocated m
emory in use for the current vector and revert bac
k to the input state
p : print; print the current value of the vecto

r
h   :   help; print a summary of the commands and o
perations possible
a value :   append; extend the vector with the addi
tional floating-point value specified
+ value :   scalar plus; add the specified floating
-point value to each element in the vector
- value :   scalar minus; subtract the specified f
loating-point value from each element in the vecto
r
* value :   scalar multiply; multiply each element
 in the vector by the specified floating-point val
ue
/ value :   scalar divide; divide each element in
the vector by the specified floating-point value
e   :   end; free up the dynamically-allocated mem
ory in use for the current vector and terminate ex
ecution
vecalc uses dynamically-allocated memory for the v
ector of floating-point values. The vector is allo
cated using new and de-allocated using delete on t
he "c", "e", or "q" commands. An "a" requires a co
mbination of allocating a new vector, copying the
contents of the old vector, adding the additional
element, and then de-allocating the old vector.

In terms of interaction with a user, the vecalc pro
gram is reasonably robust. For instance:

it terminates gracefully if the user gives an end-
of-file when the program is seeking input;
it checks if the value given in a "/" command is z
ero, and takes reasonable action if it is;
it tolerates reasonable amounts of white space (in
cluding no white space) between the operator and o
perand in the case of "+", "-", "*", and "/";
it takes a reasonable course of action if the user
 provides only white space or a null line when inp
ut is sought.
For obtaining numeric values from the user, the pro
gram may

get input using fgets(3), and convert it using atoi (3), atof(3), or sscanf(3), or
use scanf(3) or fscanf(3)
providing that the semantics described above are achieved.
The file sample_run_log.txt which accompanies this assignment specification is a sample log of a vecalc program in operation. Note that adequate end-to-end testing would go beyond the cases shown in sample_run_log.txt.

Assignment Requirements

The focus of this assignment is on the test-driven development (TDD) model discussed in class. As such, you are expected to iteratively add unit tests and code to pass those unit tests, adding no more code than is minimally required to pass your unit tests. A significant portion of the grade for your assignment will be on following the TDD process, the coverage of your unit tests, the design of your data types, and adherence to the programming guidelines given in class. Also significant will be programming style and proper documentation of your code.

For this assignment you must implement the following data types, functions, and unit tests for the functions in a single file called vecalc.cc. In subsequent assignments more of the final, intended functionality of the program will be realized.

Data Types

You must implement the following data types.

Elem â€“ for a vector element.

Vector â€“ a structured type for vectors. This structure contains an unsigned integer field for recording the size of the vector, plus a pointer to a dynamically allocated array containing the vector

elements.

Any and all types supporting the two above that you think are required.

Recall that declaring classes is not allowed. Use the standard C typedef construct for giving meaningful type names to built-in types that are used.

Supporting Functions

You must implement unit tests for the following functions, plus functions that will pass those unit tests. The function name is designed to be suggestive of the module's functionality. In a few cases, an additional statement is given to help explain the intended functionality. A partial function header is given sufficient to describe the input and return data types. Any functions with return type of Vector * return NULL on error.

bool print_vec(Vector *)

Vector *alloc_vec(void) â€” allocate an empty (zero-length) vector

void dealloc_vec(Vector *)

Vector *extend_vec(Vector *, Elem) â€” allocate a new vector one element greater in size than the input vector, copy the elements in the input vector to the new one, add the new element to the end of the new vector, and return a pointer to the new vector. The input vector is not modified.

Vector *scalar_plus(Vector *, Elem)

You should make use of assertions in your code for the above functions to catch possible logical oversights and to identify cases where preconditions or postconditions are not met. Use of such assertions are likely to aid in debugging and mean that y

our modifications to the code in the next assignment satisfied the requirements in this assignment specification (which you may otherwise forget or overlook).

You will also need a function main() which invokes the unit tests. Those unit tests will call the functions mentioned above. The unit tests are to be within a conditionally-compiled block controlled by a "#ifdef TESTING" statement. That is, the unit test code will begin with

#ifdef TESTING
and end with
#endif // TESTING
In addition you need to implement stub routines for the following functions. These stubs are to be commented, but must return an arbitrary value (NULL in this case) so that the corresponding unit tests will fail.

Vector *scalar_minus(Vector *, Elem)

Vector *scalar_mult(Vector *, Elem)

Vector *scalar_div(Vector *, Elem)

Unit tests of the above stub routines must also be present in main() within a conditionally-compiled block controlled by the symbol TESTING.

Eventually (i.e. in Assignment 3), the functions implementing the scalar operations (scalar_minus(), scalar_mult(), and scalar_div()) will be added to your vecalc program. The functions will operate "in place" on the vector passed as the argument to the function. The return value of the function will indicate whether the operation was a success. The function will return the input vector pointer on success, or NULL on error. The functions will be atomic in that the scalar operation will be performed on all the elements of the vector (if the retu

rn value indicates success), or on none of them (i
f the return value indicates failure). That is, if
 the function does not succeed, then the input vec
tor will be unmodified. However, this will all com
e in Assignment 3. In this assignment, only "stubs
" are present for the above three functions.

Finally, code for the following function is alread
y provided for you. You do not need to do anything
 to/with this code in this Assignment. However, yo
u may copy it into your vecalc.cc file if you wish.

void usage(void)

Compilation Instructions

Your program must compile, without errors or warnin
gs, with the commands

```
g++ -Wall -Wextra -o vecalc vecalc.cc
g++ -Wall -Wextra -DTESTING -o vecalc.testing vecal
c.cc
```

on tuxworld. The exceptions are that warnings abou
t the two arguments to main() being unused, and wa
rnings about unused variables occurring in program
 stubs, are permissible.

Execution

After compilation, you must have two executable pr
ograms, vecalc with no unit tests and vecalc.testi
ng containing the unit tests. Running vecalc shoul
d produce no output, and the program must terminat
e with success. Running vecalc.testing, however, m
ust execute all of your unit tests and print nothi
ng if a unit test passes, with the exception of ou
tput from print_vec(). The successful unit test fo
r print_vec() may generate output. Once execution
reaches the unit tests for scalar_minus(), scalar_
mult(), and scalar_div() the first unit test will
fail, output will be produced, and the program wil
l terminate.

The marker will be confirming the above behaviour with your submitted vecalc.cc file.

Documentation

Internal documentation must be present throughout your vecalc.cc source file. Make sure to follow the programming guidelines given in class, including the documentation guidelines involving function headers, declaration of datatypes, etc. Every function but trivial ones must include specifications that describe what the function does, preconditions, inputs (formal parameters and any accessed global variables), postconditions, return value (if any), and any effects of the function (e.g. assignment to global variables or through pointers).

Also prepare external documentation in the style of a "programmer's manual" or "reference manual" describing the program as developed to this intermediate stage. Remember that this external documentation will be helping the marker understand what you have completed. If the marker is confused and cannot understand what you have done, your grade can suffer accordingly.

The external documentation may be in any of the following forms: (plain) text, RTF, HTML, or PDF. Other types of files, included MS Word files, are not acceptable.

Use an appropriate name for the file containing your external documentation. Make sure your name, student number, and NSID appears at the beginning of the file.

Notes

You can remove warnings about argv and argc in your main function being unused parameters by changing the declaration of main to

int main(int, char**)
All lines in your vecalc.cc file are to be no more than 80 columns wide. Also remember the class material material on not intermixing tabs and spaces when formatting with white space. The LINUX/UNIX command expand(1) might be useful in this respect.

Be aware that some decimal numbers are not represented exactly as floating-point numbers. You can see examples of this in the sample run log. As another example, the following program will abort because the assertion fails

```
#include
main() {
    float f=3.14;
    assert( 3.14 == f );
}
```

while this one will not

```
#include
main() {
    float g=314.0;
    assert( 314.0 == g );
}
```

Therefore, if you use unit tests which require testing for equality of floating-point values, you are allowed to use only exactly represented values (such as 314.0 in the example above). Fortunately, there are many values that are exactly represented.
If an assert() macro causes your program to abort, you may get a "core dump" created. This is a file with a name of the form core.PID on LINUX, where PID was the PID of the process that executed an abort(). These files can be used for post-mortem debugging. However, they can also clutter up your directories. You may wish to "garbage collect" (delete) them.

Submission Instructions

All of your code must be fully contained in a sing
le file called vecalc.cc. At the beginning of this
 file include your name, student number, and NSID
in a comment block.

Upload your vecalc.cc file and documentation throug
h the moodle pages for the course.

Luxuriantly hand-crafted from only the finest HTML
tags by ...
kusalik @ cs (.usask.ca)