CSC 352, Fall 2015
Assignment 12
Due: Wednesday, December 9 at 23:59:59

Make a `Makefile` symlink!

We'll be using `make` to do compilation, so make a `Makefile` symlink just as you did for assignment 11:

```
ln -s a12/Makefile .
```

The Usual Stuff

All the usual stuff on assignments applies to this one, too: make an `a`*`N`* symlink, use the Tester, the Tester runs `make` first, etc. Refer to previous write-ups if you need a refresher.

No restrictions, but...

There are no restrictions on your solutions for this assignment; you're free to use any elements of the C language and library routines that you wish. However, the assignment is written with the intention of being solvable using only what you've seen on the slides, the previous assignments, and additional routines mentioned in this write-up. If you find yourself going beyond that set of things, you're probably overlooking a simpler solution, and/or making a problem harder than intended. And, you might not be getting practice with the things I'd like you to be getting practice with.

Probably more tests coming

A set of tests for this assignment is in place but I'd like to improve it. I'm giving myself through 3:00pm on Wednesday, December 2 to do so.

Problem 1. (12 points) `lastmod.c`

For this problem you are to write a C program named `lastmod` that prints how many minutes ago each of the files named on the command line was modified. Files are shown in order by time, with the most recently modified file last. Here's an example:

```
% lastmod *.c . a.out /dev/null / a12
/dev/null: 88652 minutes ago
/: 8231 minutes ago
vector.c: 275 minutes ago
a12: 206 minutes ago
vm.c: 204 minutes ago
lastmod0.c: 132 minutes ago
lastmod.c: 43 minutes ago
a.out: 18 minutes ago
mytime.c: 13 minutes ago
kd.c: 5 minutes ago
.: 3 minutes ago
```

Times are always shown in minutes. Note that "files" is in the broad sense—the above example includes regular files, directories, a device, and a symbolic link.

<u>Note that because the output is based on current time, file modification times, and directory contents, you'll see different results if you run the above example.</u>

If a non-existent file is named, `lastmod` uses `perror(3)` to print the name and then exits immediately

```
% lastmod no-such-file *.c
no-such-file: No such file or directory
```

If run with no arguments, `lastmod` produces no output.

Because of its nature, `lastmod` is problematic to test with the tester. There are only two simple tests for `lastmod`; **the usual guarantee of 75% of the points for passing all the tester tests does not apply to `lastmod`.** Do some testing on your own and compare your results to results from `a12/lastmod`.

Note that the tester creates/touches three files in your directory: `x`, `y`, and `z`.

Implementation notes

Use `time(3)` to get the current time. Use `stat(2)` to get the last modification times for the files. I sort with `qsort(3)`.

Remember that the shell does wildcard expansion—you don't need any code to handle an argument like `*.c`.

Problem 2. (12 points) `kd.c`

"`kd`" stands for "keep/discard". `kd` "filters" lines read on standard input, keeping or deleting lines based on command line specifications. The following invocation says to keep lines that contain an "`e`":

```
% cal -h 12 2015 | kd e
   December 2015
Su Mo Tu We Th Fr Sa
```

If an argument starts with a minus, it indicates to discard lines that contain the text following the minus. Example:

```
% cal -h 12 2015 | kd -em e
Su Mo Tu We Th Fr Sa
```

Think of **`kd -em e`** as a two step process: (1) Discard lines that contain "**`em`**". (2) Of the lines that remain, keep only the lines that contain "**`e`**".

Here are two examples that demonstrate that the order of the arguments is inconsequential:

```
% seq 1000 | kd 00 -3 -5 -7 -000
100
200
400
600
800
900
% seq 1000 | kd -7 00 -000 -5 -3
100
200
400
600
800
900
```

`kd` accepts any number of arguments of any length. If run with no arguments, `kd` produces a usage message:

```
% kd
Usage: kd [-]STR1 [-]STR2 ... [-]STRN
```

Important: For this problem I want you to use a "hybrid" solution, based on either `popen(3)` or `system(3)`. As a first step, note that **`kd -7 00 -000 -5 -3`** is equivalent to a pipeline of `fgrep` commands:

```
% seq 1000 | fgrep -v 7 | fgrep 00 | fgrep -v 000 | fgrep -v 5 | fgrep
-v 3
100
200
400
600
800
900
```

In fact, `kd` is a simple program: it builds a pipeline of `fgrep` commands and runs it!

I'd like you to both think and experiment a little to determine if `system(3)` is sufficient or if `popen(3)` is needed, but note that the `command` shown in the man page synopses for `system` and `popen` can be a pipeline, not just a single command.

Problem 3. (40 points) `vector.c`

A simple but useful data structure is an expandable array, like Java's `ArrayList`. In this problem you are to implement a generalized set of functions that implement an expandable array. This expandable array will be called a *vector*. The design of the interface, described below, is strongly based on the ideas in *C Interfaces and Implementations*, by David R. Hanson—an excellent book by a great teacher[1].

Code that uses our vector data structure will work with vectors using *opaque pointers*. To help you understand the merit of using opaque pointers, I'll start by talking about Java.

A great feature of Java is that the fields of a class can be declared as private, preventing code in other classes from using those fields directly, and thus becoming dependent on their presence. The implementor of a class is free to completely change the set of private fields; all that matters is that the interface provided by the public methods doesn't change. For example, a `Rectangle` class might have public methods such as `getWidth()`, `getHeight()`, and `getArea()`. The implementor of `Rectangle` might choose to support those methods with two private fields: `width` and `height`.

Over time it may turn out that `getArea()` is called very frequently but `getWidth()` and `getHeight()` are rarely used. To speed up `getArea()` we might choose to replace the private fields `width` and `height` with two different fields, `area` and `aspectRatio`, and compute those in a `Rectangle(double width, double height)` constructor. Because `width` and `height` were private, there's no code outside the `Rectangle` class using `width` and `height`, and there will be no fuss, mail frenzies, or meetings about replacing them with `area` and `aspectRatio`, but users will find that `getArea()` now runs a little faster. (Yes, `getWidth()` and `getHeight()` will be slower because they'll be computing the width and height based on the area and aspect

[1]Many years ago I had the privilege of taking CSC 453 here with Dr. Hanson. In the course of the semester, we wrote a compiler for a subset of C, and then a simple linker and a debugger, too. Hanson wrote the DEC-10 C compiler and library that we used to write our programs, sometimes getting features in place just a day or two before we needed them. When somebody asks me, "What's the best course you ever had?", Hanson's 453 is my answer. Hanson and a colleague, Chris Fraser, wrote *A Retargetable C Compiler: Design and Implementation*, another excellent book.

ratio, but we'll imagine it was important to not increase the memory required for a `Rectangle`.)

There's no ability to hide the members of a C `struct`, but we can hide the `struct` itself! We see a mild form of this with `FILE`: we get a `FILE *` from `fopen` but we know that we shouldn't write code that depends on the members of `FILE` because they might change in the next version of the library. We then pass those `FILE *` values to routines like `fprintf`, `fread`, and `fseek` to perform operations on the files they represent.

We take a much stronger approach to hiding the implementation of vector. Here's the only datatype a user of vector can count on:

```
typedef struct Vector *Vector_T;
```

This `typedef` declares `Vector_T` to a pointer to a `struct Vector` but we never reveal the `struct` itself to the user. We can say that `Vector_T` is an *opaque pointer*—the user of `Vector_T` has no idea what the members of `struct Vector` are. This gives us complete freedom to change `struct Vector` whenever we wish, without breaking any code that uses our vector functions.

Here's the structure I currently use to represent a vector:

```
typedef struct Vector {
    int cell_size;       // # of bytes in a cell
    int active_cells;    // # of cells with value
    int allocated_cells; // # of cells that space has been
                         //   allocated for
    void *cells;         // Address of first byte of first cell
    void *cells_end;     // Address of first byte past last cell
    } *Vector_T;
```

Your `struct Vector` might be entirely different, and I might change mine tomorrow, but to give us stable ground for discussion, we'll use my implementation as an example to reference.

A vector consists of some number of "cells", all of which are the same size (`cell_size`). A vector that holds `char` values will have a `cell_size` of 1; a vector that holds `double` values will have a cell size of 8. When a vector is first created, I allocate space for four cells, but you might do something different. The member `cells` holds the address of the first byte of the first cell.

The `Vector_new` function is used to create a vector. Here is its prototype:

```
Vector_T Vector_new(int cell_size);
```

The end result of `Vector_new` is that a `Vector` structure is allocated using `malloc` or `calloc`, the structure is initialized, and the address of that structure is returned by `Vector_new`. If the allocation fails, `Vector_new` returns zero.

A vector of `char` values can be made like this:

```
Vector_T letters = Vector_new(sizeof(char));
```

A vector of `int` values can be made like this:

```
Vector_T ints = Vector_new(sizeof(int));
```

Values are appended to a vector with `Vector_put`:

```
Vector_T Vector_put(Vector_T v, void *value);
```

`Vector_put`'s first argument is a `Vector_T` (a pointer) produced by `Vector_new`. The second argument is the address of a value to add. `cell_size` bytes are copied from `value` to the next cell of the vector. The addition may cause an expansion. My current implementation doubles the number of allocated cells when an expansion is required; yours might do something different.

Here's a loop that adds `char` values to `letters`:

```
for (char *p = "testing"; *p; p++)
    Vector_put(letters, p);
```

The function `int Vector_length(Vector_T v)` returns the number of values the vector is currently holding.

```
int len = Vector_length(letters);  // len is 7
```

The function

```
void Vector_print(Vector_T v, FILE *fp,
       void (*pfcn)(FILE *, void *p))
```

outputs the contents of the vector to `fp`. `Vector_print` calls the function referenced by `pfcn` with the address of each cell in turn, along with the `FILE *` passed to `Vector_print`. Example:

```
Vector_print(letters, stdout, print_char);
fprintf(stdout, "\n");
```

Output:

```
[t,e,s,t,i,n,g]
```

Note that the sequence of values is surrounded with square brackets; commas separate values. `Vector_print` does not output a trailing newline.

Here is the `print_char` function used by `Vector_print` above:

```
void print_char(FILE* fp, void *p)
{
    fputc(*(char*)p, fp);
}
```

The printing function can do whatever it wants. Here's a more verbose `print_char`:

```
void print_char(FILE* fp, void *p)
{
    fprintf(fp, "'%c'(%d)", *(char*)p, *(char*)p);
}
```

Instead of `[t,e,s,t,i,n,g]`, it would produce this:

```
['t'(116),'e'(101),'s'(115),'t'(116),'i'(105),'n'(110),'g'(103)]
```

A value can be retrieved with `Vector_get`:

```
int Vector_get(Vector_T v, int n, void *result)
```

It is important to understand that **`Vector_get` does not return the value that is retrieved**. Instead, it copies the `n`th value (zero-based) into the memory pointed to by `result` and returns 1. If `n` is out of bounds, `Vector_get` returns 0 and the memory referenced by `result` is unchanged.

The following loop prints the elements of `letters` in reverse order:

```
int len = Vector_length(letters);
char c;

for (int i = len-1; Vector_get(letters, i, &c); i--)
    printf("%d: %c\n", i, c);
```

Output:

```
6: g
5: n
4: i
3: t
2: s
1: e
0: t
```

You may wonder why `Vector_get` doesn't return a value. It is not possible to write a C function whose return value varies in size. Therefore, to accommodate values of any size, `Vector_get` must resort to copying the requested value into a buffer.

A value in a vector can be changed with `Vector_set`:

```
int Vector_set(Vector_T v, int n, void *value)
```

`Vector_set` copies bytes from `value` into the `n`th cell of the vector, which must already exist.

Here's a loop that uses `Vector_get` and `Vector_set` to capitalize characters in the vector `letters`:

```
for (int i = 0; Vector_get(letters, i, &c); i++) { // c is a char
    c = toupper(c);
    Vector_set(letters, i, &c);
    }
```

If `n` is out of range, `Vector_set` returns 0. It returns 1 otherwise.

The value at a specified location can be deleted with `Vector_delete`:

```
int Vector_delete(Vector_T v, int n);
```

The following code deletes the first and last values in the vector `ints`:

```
Vector_delete(ints, 0);
Vector_delete(ints, Vector_length(ints)-1);
```

If `n` is out of range, `Vector_delete` doesn't change the vector and returns 0. It returns 1 otherwise.

`Vector_delete_if` applies a *predicate* to each value in the vector and deletes the elements that satisfy the predicate. Here's the prototype:

```
void Vector_delete_if(Vector_T v, int (*pred)(void *p));
```

`pred` is a pointer to a function that expects the address of a value. If the value satisfies a condition of some sort, the value is deleted from the vector. We can say that deletion is *predicated* on that condition.

Here's a predicate that's satisfied if a value is an odd number:

```
int is_odd_int(void *vp)
{
    return *(int*)vp % 2 == 1;
}
```

Here's a call that uses `is_odd_int` to remove odd integers from the vector `ints`:

```
Vector_delete_if(ints, is_odd_int);
```

Here's a hint for implementing `Vector_delete_if`: starting with the last value working toward the first will avoid some bookkeeping headaches. Also, my implementation of `Vector_delete_if` uses `Vector_delete`.

A copy of all the values stored in a vector can be obtained with `Vector_alloc`:

```
void *Vector_alloc(Vector_T v, void *end_value);
```

`Vector_alloc` uses `malloc` to allocate a properly sized array and copies `v`'s values into it. If `end_value` is non-zero, an extra, final cell is allocated in the result array, and the value pointed to by `end_value` is copied to that cell.

Here's an example that uses `Vector_alloc` to create a C string from the characters in `letters`:

```
char NUL = 0;
char *s = Vector_alloc(letters, &NUL);
printf("s = '%s'\n", s);
```

The block of memory returned by `Vector_alloc` above is eight bytes long—seven for `"testing"` and one for the ending value. If instead `Vector_alloc(letters, 0)` is used, a seven-byte block is produced, and `printf` cannot be reliably used because a terminating zero is not present. The caller of `Vector_alloc` is expected to use `free(3)` when the block returned by `Vector_alloc` is no longer required.

Finally, `Vector_free` is used to free the memory associated with a vector:

```
void Vector_free(Vector_T *vp);
```

Note the type of `vp`: a pointer to a `Vector_T`, which is in turn a pointer to a `Vector` structure. It is used like this:

```
Vector_free(&letters);  // Note: &letters
```

`Vector_free` not only frees the memory associated with the vector; **it also zeroes the pointer!** When `Vector_free` returns, the value of `letters` is 0. The underlying idea is that zeroing the pointer prevents the pointer from being used to inadvertently commit a used-after-freed error. A subsequent call using `letters`, like this,

```
        len = Vector_length(letters);
```

is guaranteed to produce a fault.

`a12/vector.h` is a header file with a `typedef` for `Vector_T` and documented prototypes for all the functions you are to write.

Here's how my `vector.c` starts:

```
        #include <string.h>
        #include <stdlib.h>
        #include "/cs/www/classes/cs352/fall15/a12/vector.h"

        typedef struct Vector {
            int cell_size;         // # of bytes in a cell
            int active_cells;      // # of cells with value
            ...
            } *Vector_T;

        Vector_T Vector_new(int cell_size)
        {
        ...
        }

        ...
```

As you work on vector, keep in mind the title of Hanson's book: *C Interfaces and Implementations*. `vector.h` defines the interface. `vector.c` is the implementation.

Problem 4. (55 points) `vm.c`

Introduction

In this problem you are to implement a portion of a simple "virtual machine". One type of virtual machine is that provided by applications like VMWare and VirtualBox —software that partially or completely emulates hardware. Another type of virtual machine is exemplified by the Java Virtual Machine: the JVM is an idealized machine designed for the execution of Java programs. Its instruction set was developed with Java in mind. Most commonly a JVM is implemented in software but hardware JVMs have been developed, too.

The virtual machine you'll be working on is called "`vm`". It is not designed with any specific higher-level language in mind but it does borrow some elements from the Icon virtual machine.

My solution for `vm` is over 800 lines of code (counting comments) but over 600 of those lines are supplied in `a12/vm_starter.c`. **Your task is to understand the following documentation and existing code, and then, starting with `a12/vm_starter.c`, complete a list of "todo" items to in turn produce an implementation of `vm` that matches the following documentation.**

Here are the basics of `vm`:

- `vm` supports three data types: integer, string, and list. The list type is heterogeneous—a list can comprise any combination of integers, strings, or other lists.
- `vm` is "stack-based"—the operands for instructions are on the stack. Values resulting from instructions are pushed on the stack.

- `vm` provides variables—values can be stored by name and later retrieved.

- `vm` does not provide any instructions to alter the flow of control. `vm` simply executes a "straight line" sequence of instructions and then exits. A next step with `vm` would be to add branching instructions but `vm` is big enough as-is to be a realistic and interesting C programming problem.

A `vm` tutorial

The following section shows `vm` in operation through a series of short examples. We start with a `vm` program that has four instructions:

```
% cat int1.vm
int 1
int 10
int 100
dstack
```

The instruction `int N` pushes the value `N` on the stack. The `dstack` instruction shows the contents of the stack.

The program is run by naming it as an argument of the virtual machine executable, which is `vm`. My implementation of `vm` is in `a12/vm`.

Execution of `int.vm`:

```
% vm int1.vm
--- Stack ---
002: 100
001: 10
000: 1
```

Two very simple instructions are `dup` and `swap`. The `dup` instruction duplicates the value on the top of the stack. The `swap` instruction swaps the top two elements on the stack. Example:

```
% cat dupswap1.vm
int 10
int 20
swap
dstack2 Before dup
dup
dstack2 After dup
```

```
% vm dupswap1.vm
--- Stack (Before dup) ---
001: 10
000: 20
--- Stack (After dup) ---
002: 10
001: 10
000: 20
```

The above example also illustrates the `dstack2` instruction. `dstack2` is just like `dstack`, except it labels the output with the operand of the instruction.

The `add` instruction replaces the top two elements of the stack with their sum. The `sub`, `mul`, and `div` instructions are similar, performing subtraction, multiplication, and division on the top two elements, and replacing them with the result. In each case the top of stack value is the right-hand operand. It is a fatal error if either operand is not an integer.

Here is an example that uses all four arithmetic instructions:

```
% cat math1.vm
int 3
int 10
sub
dstack
int -10
mul
dstack
int 14
dstack
div
dstack
```

```
% vm math1.vm
--- Stack ---
000: -7
--- Stack ---
000: 70
--- Stack ---
001: 14
000: 70
--- Stack ---
000: 5
```

The `str` instruction pushes a string on the stack. The literal starts after the blank following the opcode and extends to the end of the line. Example:

```
% cat str1.vm
int 1
str just a test
str   here
dstack
```

```
% vm str1.vm
--- Stack ---
002: "  here"
001: "just a test"
000: 1
```

The `list N` instruction forms a list from the top `N` elements of the stack and replaces them with the list. Example:

```
% cat list1.vm
str bottom
int 1
str two
int 3
str IV
list 4
dup
str top
dstack
list 3
dstack
```

```
% vm list1.vm
--- Stack ---
002: "top"
001: [1,"two",3,"IV"]
000: "bottom"
--- Stack ---
000: ["bottom",[1,"two",3,"IV"],"top"]
```

As mentioned earlier, lists are heterogeneous—they can contain integers, strings, and other lists that in turn may be heterogeneous.

The `concat` instruction concatenates the two strings or two lists on top of the stack and replaces them with the resulting string or list. The top of stack value becomes the rightmost elements of the result. Here is an example with strings:

```
% cat concat1.vm
str a
str  test
concat
dstack
str  here
concat
dstack
```

```
% vm concat1.vm
--- Stack ---
000: "a test"
--- Stack ---
000: "a test here"
```

Here is an example with lists:

```
% cat concat2.vm
int 10
int 20
list 2
int 30
int 40
list 2
dstack2 Ready to concat
concat
dstack2 Result
```

```
% vm concat2.vm
--- Stack (Ready to concat) ---
001: [30,40]
000: [10,20]
--- Stack (Result) ---
000: [10,20,30,40]
```

The `size` instruction replaces the string or list on top of the stack with its length (an integer). Example:

```
% cat size1.vm
str testing
size
dstack2 Size of string
dup
list 2
size
dstack2 Size of list
```

```
% vm size1.vm
--- Stack (Size of string) ---
000: 7
--- Stack (Size of list) ---
000: 2
```

The `at` instruction selects an element from a string or a list. The next-to-top value is the aggregate and the top value is an integer that specifies the value to select, zero-based. The two operands are replaced with the selected value. Here is an example with a string:

```
% cat at1.vm
string abc
int 1
at
dstack
```

```
% vm at1.vm
--- Stack ---
000: "b"
```

Here's an example with a list:

```
% cat at2.vm
str abc
int 10
list 1
list 1
str x
list 1
list 2
int 20
list 3
dstack2 Before 'at'
int 1
at
dstack2 Result of 'at'
```

```
% vm at2.vm
--- Stack (Before 'at') ---
000: ["abc",[[[10]],["x"]],20]
--- Stack (Result of 'at') ---
000: [[[10]],["x"]]
```

Note that the value selected by `at` in this case is a list of lists.

The `sort` instruction replaces the string or list on top of the stack with a sorted copy of itself. Here is an example with a string.

```
% cat sort1.vm
str virtual machine
dup
sort
dstack
```

```
% vm sort1.vm
--- Stack ---
001: " aacehiilmnrtuv"
000: "virtual machine"
```

The string is `dup`'d before it is sorted to illustrate that sorting produces a new value; it doesn't change the old one.

An example with a list follows. Note that the list is heterogeneous. The rules for sorting heterogeneous lists are specified in a per-instruction summary later in this write-up.

```
% cat sort2.vm
int 10
str two
int 2
dup
list 1
str one
list 5
dup
dstack2 Before sort
sort
dstack2 Sorted
```

```
% vm sort2.vm
--- Stack (Before sort) ---
001: [10,"two",2,[2],"one"]
000: [10,"two",2,[2],"one"]
--- Stack (Sorted) ---
001: [2,10,"one","two",[2]]
000: [10,"two",2,[2],"one"]
```

As with the string example, the aggregate is `dup`'d before the sort to emphasize that `sort` is operating on a copy of the list.

`vm` provides a minimal facility for variables: there can be a fixed number of variables and in turn, variable names have a maximum size.

The `setvar` and `getvar` instructions set and get the value of variables. `setvar name` pops the top of stack value and stores it in the variable `name`. A variable comes into existence when the first `setvar` for that name is executed.

`getvar name` pushes the value of the variable name on the stack. It is a fatal error if the variable named does not exist.

The `dvars` instruction prints the name and value of each variable, ordered by variable name.

An example with variables:

```
% cat var1.vm
int 10
setvar x
getvar x
dup
add
setvar y
dstack
dvars
```

```
% vm var1.vm
--- Stack ---
--- Variables ---
x = 10
y = 20
```

`vm` Instruction Reference

Below is a summary of `vm`'s instructions, in alphabetical order. The stack transformation performed by each instruction is shown as `<`*`OLD STACK CONTENTS`*`>` **`->`** `<`*`NEW STACK CONTENTS`*`>`. An ellipsis (**`...`**) indicates the portion of the stack that is not involved in execution of the instruction.

Any instruction-specific errors are cited. It is implicit that it is an error condition for too few operands to be present on the stack. (For example, a `dup` as the first instruction.)

`add, sub, mul, div`

Performs an arithmetic operation on the top two values on the stack:

```
   ..., VALUE1, VALUE2 -> ..., VALUE1 <add/sub/mul/div> VALUE2
```

`at`

Selects a value from an aggregate using a zero-based integer index:

```
   ..., STRING, INTEGER -> ..., STRING[INTEGER] (a string)
   ..., LIST, INTEGER -> ..., LIST[INTEGER] (a value)
```

Errors:

`INTEGER` out of range

Next-to-top value not a string or a list

`concat`

Concatenates the top two values on the stack, which must both be strings or lists:

```
  ..., STRING1, STRING2 ->
  ..., STRING having chars in STRING1, then chars in STRING2

  ..., LIST1, LIST2 ->
  ..., LIST having elements in LIST1, then elements in LIST2
```

Error: Top two values not aggregate of same type

`dstack`

Displays the contents of the stack.

`dstack2 LABEL`

Like `dstack`, but includes the specified label, to help associate a stack dump with a point in the code.

`dvars`

Displays the variables

`dup`

Duplicates the top of stack value:

```
   ..., VALUE -> ..., VALUE, VALUE
```

`getvar NAME`

Pushes the value of the named variable on the stack.

```
   ... -> ..., VALUE of NAME
```

Error: No variable with the specified name exists.

`int INTEGER`

Pushes an integer with value N on the stack:

```
      ... -> ..., INTEGER
```

`list N`
Forms a list from the top N values on the stack:
```
... VALUE1, VALUE2...VALUEN -> ..., [VALUE1, VALUE2...VALUEN]
```

`setvar NAME`
Stores the top of stack value in the named variable, creating it if necessary.
```
..., VALUE -> ...
```

`size`
Replaces the top of stack aggregate with its length:
```
..., AGGREGATE -> ..., length of AGGREGATE
```

`sort`
Replaces the top of stack aggregate (a string or list) with a sorted copy of itself.
```
..., AGGREGATE -> ..., sorted copy of AGGREGATE
```

If the aggregate is a string, then the string is sorted on a character by character basis.

If the aggregate is a list, it is sorted according to the following rules:

1. If the aggregate contains any integers, they appear first in the result, in ascending order.

2. If the aggregate contains any strings, they appear next, in ascending order, as determined by `strcmp`.

3. If the aggregate contains any lists, they appear next, ordered by the number of elements in the lists. For example, all one-element lists appear after any empty lists and before any two-element lists.

NOTE: The ordering of lists of the same length is unpredictable. For example, the list `[[1,2],[1],[3,4]]` can correctly sort to either `[[1],[1,2],[3,4]]` or `[[1],[3,4], [1,2]]`.

`str STRING`
Pushes a string with value `STRING` on the stack:
```
... -> ..., STRING
```

`swap`
Swaps the top two values on the stack:
```
..., VALUE1, VALUE2 -> ..., VALUE2, VALUE1
```

The TODO list

The purpose of all of the previous material is to help get you to the point where you are able to understand the code in `a12/vm_starter.c`.

Here are your tasks for this problem:

- Add code to `load_program()` to cause a `"//"` to be treated as a comment to end of line, just like in Java.

- Add entries to the `Opcode enum` for `div`, `dstack2`, `mul`, `size`, `sub`, and `swap`. Add entries to `opmap` for those same instructions.

- The implementation of `dstack` and `dstack2` is incomplete. All that's present is a case in `execute()`

for `OP_DSTACK`. It calls `dump_stack`, which you are to implement. Also add a case in `execute()` for `OP_DSTACK2`.

- Add cases to `execute()` for `OP_MUL`, `OP_SUB`, and `OP_DIV`. Note that they will be very similar to `OP_ADD`.

- Add a case to `execute()` for `OP_SWAP`.

- Add a case to `execute()` for `OP_SIZE`. Looking at `OP_INT` may help with this one.

- Implement `getvar()` and `dump_vars()`, called from the `OP_GETVAR` and `OP_DVARS` cases in `execute()`.

- The `OP_AT` case in `execute()` calls two routines, `String_at` and `List_at`, that are not implemented. Implement them.

- The `OP_CONCAT` case in `execute()` calls `List_concat`, which is not implemented. Implement it.

- The `OP_SORT` case in `execute()` calls `List_sort`, which is not implemented. Implement it.
  Hint: Use `qsort`, of course. On each call the comparison routine will need to see if the values are of the same type and if so, compare them directly. If the types differ, call the supplied `type_sequence` routine to get a sequence number for each type. Then, simply compare the sequence numbers to see which one is less than the other.

Miscellaneous

I see some pros and cons to this problem. I think that what's good about it is that virtual machines such as this are a class of applications that are almost always written in C or C++, and are one of the things that C is best at. I also think you'll have fun adding pieces to the starter code and seeing the whole thing gradually come to life. What worries me about it is that it has a high conceptual load—I think there's a risk that some students may have trouble seeing the big picture on this problem, and then recognizing that the picture isn't that big after all.

Probably the first thing you should do code-wise is to copy `a12/vm_starter.c` to `vm.c` in your own directory and try to build `vm` with `make`. You'll should something like this, assuming you've made a symlink to `a12/Makefile`.

```
% cp a12/vm_starter.c vm.c
% make vm
gcc -Werror -Wall -g -std=gnu1x -I/cs/www/classes/cs352/fall15/h -o vm
vm.c
/tmp/ccQrnTdT.o: In function `execute':
/home/whm/352/a12/work/vm.c:511: undefined reference to `dump_stack'
/home/whm/352/a12/work/vm.c:543: undefined reference to `List_concat'
/home/whm/352/a12/work/vm.c:555: undefined reference to `List_sort'
/home/whm/352/a12/work/vm.c:563: undefined reference to `String_at'
/home/whm/352/a12/work/vm.c:565: undefined reference to `List_at'
/home/whm/352/a12/work/vm.c:572: undefined reference to `getvar'
/home/whm/352/a12/work/vm.c:582: undefined reference to `dump_vars'
collect2: ld returned 1 exit status
make: *** [vm] Error 1
```

Start by "stubbing out" those missing functions with implementations that just print "`Oops! List_at called!`" (for example). Then, implement simple instructions like `dstack`, `dstack2`, and `swap`.

Problem 5. Extra Credit **`observations.txt`**

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a12/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N.YYYYMMDD.HHMMSS`*`.tz` You can run `a12/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `352f15` if you need to recover a file—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a12/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a12/delivs)
   43   113   995 lastmod.c
   38   114   811 kd.c
  131   380  3177 vector.c
  814  2328 20023 vm.c
 1026  2935 25006 total

% for i in $(grep -v txt a12/delivs); do echo $i: $(tr -dc \; < $i | wc
-c); done
lastmod.c: 19
kd.c: 20
vector.c: 56
vm.c: 284
```

Remember that over 600 of those lines in `vm.c` come from `a12/vm_starter.c.`

Aside from `vm.c`, my code has few comments.

Miscellaneous

This assignment is based on the material on C slides 1-517.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them, has completed all previous assignments, knows how to use `gdb` and `valgrind`, and has done the required reading will need 12-16 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems.

If you put ten hours into this assignment and don't seem to be close to completing it, it's probably time to touch base with us. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)