GritVM Interpreter

COP 4530 Programming Project 2

Instructions

For Programming Project 2, you will be implementing an interpreter for a programming language called GritVM. Your GritVM will read in a file of code written in this specific GVM language, run the instructions, and be able to return the results held in the GritVM object's memory. The GritVM has two variables that contain its memory: a NodeList that contains the instructions and a Vector that contains the data. There will also be a single variable called the accumulator. The accumulator stores the results of various operations as an intermediate. The accumulator is an implicit operand for all mathematical calculations.

Below is an example of some of your private members for the GritVM.

```
std::vector<long> dataMem;
std::list<Instruction> instructMem;
std::list<Instruction>::iterator currentInstruct;
STATUS machineStatus;
long accumulator;
```

For this project. I am allowing you to use the C++ Standard Template Library version of the Vector and List. You must use these STL data types for your GritVM. The GritVM can only operate on long data types, so that is the only data type you will need to worry about for your machine data.

You will be responsible for programming in the functionality of the following instructions in your implementation of GritVM. Notice that the memory management functions match the Vector ADT nearly identically. DM represents data memory:

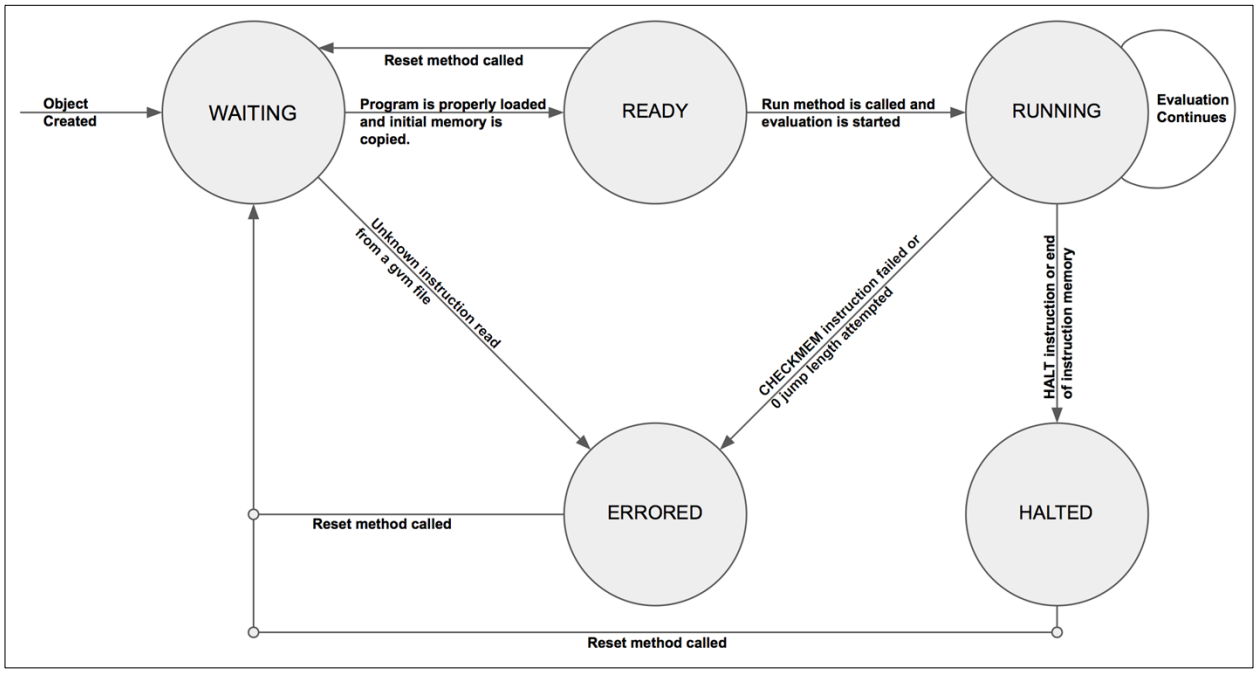
Instruction	Type	Call	Notes
CLEAR	Accumulator Functions	CLEAR	Set the accumulator to 0
AT	DM Management Functions	AT X	Sets the accumulator to the value at DM[X] A = DM[X]
SET	DM Management Functions	SET X	Sets the DM[X] to the value in the accumulator DM[X] = A
INSERT	DM Management Functions	INSERT X	Inserts in DM at location X the value in the accumulator

Instruction	Type	Call	Notes
ERASE	DM Management Functions	ERASE X	Erases location X in the DM
ADDCONST	Accumulator Maths with a Constant	ADDCONST C	Adds C to the accumulator value A = A + C
SUBCONST	Accumulator Maths with a Constant	SUBCONST C	Subtracts C from the accumulator A = A – C
MULCONST	Accumulator Maths with a Constant	MULCONST C	Multiplies C to the accumulator value A = A * C
DIVCONST	Accumulator Maths with a Constant	DIVCONST C	Divides C from the accumulator value A = A / C
ADDMEM	Accumulator Maths with a Memory Location	ADDMEM X	Adds DM[X] to the accumulator value A = A + DM[X]
SUBMEM	Accumulator Maths with a Memory Location	SUBMEM X	Subtracts DM[X] from the accumulator A = A – DM[X]
MULMEM	Accumulator Maths with a Memory Location	MULMEM X	Multiplies DM[X] to the accumulator value A = A * DM[X]
DIVMEM	Accumulator Maths with a Memory Location	DIVMEM X	Divides DM[X] from the accumulator value A = A / DM[X]
JUMPREL	Instruction Jump Functions	JUMPREL Y	Goes forward/back Y instructions from the current instruction (can be negative)
JUMPZERO	Instruction Jump Functions	JUMPZERO Y	Goes forward/back Y instructions from the current instruction (can be negative) if accumulator is 0. Otherwise just move forward 1 from the current instruction.
JUMPNZERO	Instruction Jump Functions	JUMPNZERO Y	Goes forward/back Y instructions from the current instruction (can be negative) if accumulator is not 0. Otherwise just move forward 1 from the current instruction.
NOOP	Misc Functions	NOOP	Perform no operation. Counts as an instruction but does nothing.
HALT	Misc Functions	HALT	Stop the GritVM and switch status to HALTED
OUTPUT	Misc Functions	OUTPUT	Output accumulator to std::cout
CHECKMEM	Misc Functions	CHECKMEM Z	Checks to make sure DM is of at least size Z. If not, switch status to ERRORED

The GritVM object can be in multiple states during its object lifetime. The table below shows these statuses:

Status	Meaning
WAITING	Waiting to load a program into instruction memory
READY	A program has been loaded into instruction memory and is ready to run
RUNNING	The machine is actively running a program
HALTED	The program halted the machine whether by a HALT instruction or reaching the end of the instructions
ERRORED	The machine stopped because of an error in the program
UNKNOWN	Unknown status. Should never happen in normal control flow.

The following image demonstrates how the GritVM moves between states.



The primary flow of the machine should be as follows:

- When created, the GritVM sets the accumulator to 0, and it's status to WAITING

- A program is loaded into the GritVM object by filename (by the load method)
- If the machine status is anything other than WAITING, return the current status
  - If the GritVM cannot read a file, the method throws an exception
- Each line of the file is read in and parsed into its proper instruction and argument
  - If the instruction is not recognized, change the machine status to ERRORED and return
  - If the line is blank or starts with a #, then skip that line (it is a comment or white space)
- Add that instruction to the instructMem list

    - If the instructMem size is 0 set the status to WAITING,
        - Otherwise, set it to READY
- Copy the vector passed into the load method to the dataMem vector
- Return the current status

- If a GritVM object is READY receiving a call to the run method, then the instructions can be evaluated
    - Otherwise, return the current status
- Evaluate the current instruction
- After evaluation, move the current instruction forward the proper amount
    - 1 for regular instructions
    - Y for jumps if a jump is triggered (where Y != 0, if Y is 0, set status to ERRORED and return run method)
    - Set status to HALTED if the last instruction was HALT or the last instruction has been reached
- Return the current status

Abstract Class Methods

There will be an abstract class provided for your GritVM class to inherit from. The following methods must be defined:

`STATUS load(const std::string filename, const std::vector<long> &initialMemory)`

This method loads the GVM program into the GritVM object and copies initialMemory into the data memory. Returns the current machine status. Throws if file cannot be read.

`STATUS run()`

This method starts the evaluation of the instructions. Returns the current machine status.

`std::vector<long> getDataMem()`

Returns a copy of the current dataMem

`STATUS reset()`

Sets the accumulator to 0, clears the dataMem and instructMem, sets the machineStatus to WAITING.

Other things in GritVMBase hpp/cpp

Also provided in the base class files are the enums that define the various stats and instructions for the GritVM. There is an instruction struct for holding an instruction, and it's argument. There are also five helper methods for you. All of these methods are in the namespace GVMHelper.

`std::string statusToString(STATUS s);`

Converts a status value to a string version of that status.

`STATUS stringToStatus(std::string s);`
Converts a string of a status value to an enum of that status.

`std::string instructionToString(INSTRUCTION_SET s);`
Converts an instruction value to a string version of that instruction.

`INSTRUCTION_SET stringtoInstruction(std::string s);`
Converts a string of an instruction value to an enum of that instruction.

`Instruction parseInstruction(std::string GVMLine);`
Given a line of a GVM file, this function returns an Instruction struct containing the instruction enum and the long argument (if applicable)

Examples

Below are some examples of how your code should run

```
GritVM vm;
// Status: WAITING
// Accumulator: 0
// *** Data Memory ***
// *** Instruction Memory ***

vm.load("altseq.GVM", inMem);
// Status: READY
// Accumulator: 0
// *** Data Memory ***
// Location 0: 15
// *** Instruction Memory ***
// Instruction 0: CHECKMEM 1
// Instruction 1: CLEAR 0
// Instruction 2: ADDCONST 1
// Instruction 3: INSERT 1
// Instruction 4: INSERT 2
// Instruction 5: AT 0
// Instruction 6: ADDCONST 1
// Instruction 7: SUBMEM 2
// Instruction 8: JUMPNZERO 2
// Instruction 9: JUMPZERO 9
// Instruction 10: CLEAR 0
// Instruction 11: AT 1
// Instruction 12: MULCONST -2
// Instruction 13: SET 1
// Instruction 14: AT 2
// Instruction 15: ADDCONST 1
// Instruction 16: SET 2
```

```
// Instruction 17: JUMPREL -12
// Instruction 18: AT 1
// Instruction 19: MULCONST 3
// Instruction 20: ADDCONST 4
// Instruction 21: SET 1
// Instruction 22: HALT 0

vm.run();
// Status: HALTED
// Accumulator: -98300
// *** Data Memory ***
// Location 0: 15
// Location 1: -98300
// Location 2: 16
// *** Instruction Memory ***
// Instruction 0: CHECKMEM 1
// Instruction 1: CLEAR 0
// Instruction 2: ADDCONST 1
// Instruction 3: INSERT 1
// Instruction 4: INSERT 2
// Instruction 5: AT 0
// Instruction 6: ADDCONST 1
// Instruction 7: SUBMEM 2
// Instruction 8: JUMPNZERO 2
// Instruction 9: JUMPZERO 9
// Instruction 10: CLEAR 0
// Instruction 11: AT 1
// Instruction 12: MULCONST -2
// Instruction 13: SET 1
// Instruction 14: AT 2
// Instruction 15: ADDCONST 1
// Instruction 16: SET 2
// Instruction 17: JUMPREL -12
// Instruction 18: AT 1
// Instruction 19: MULCONST 3
// Instruction 20: ADDCONST 4
// Instruction 21: SET 1
// Instruction 22: HALT 0

std::vector<long> outmem = vm.getDataMem();

vm.reset();
// Status: WAITING
// Accumulator: 0
// *** Data Memory ***
```

```
// *** Instruction Memory ***
```

Deliverables

Please submit complete projects as zipped folders. The zipped folder should contain:

- GritVM.cpp (Your written GritVM class)
- GritVM.hpp (Your written GritVM class)
- GritVMBase.cpp (The provided base class, enums, structs, and helper functions)
- GritVMBase.hpp (The provided base class, enums, structs, and helper functions)
- PP3Test.cpp (Test file)
- catch.hpp (Catch2 Header)
- altseq.gvm (A Alternating Sequence Program for GritVM)
- sumn.gvm (A Summation Program for GritVM)
- surfarea.gvm (A Surface Area Program for GritVM)
- test.gvm (A Test Program for GritVM)

And any additional source and header files needed for your project.

Hints

Remember, the math can only be done to the accumulator. You will never be doing the four functions on data memory locations. For example, the SUBMEM instruction should perform accumulator = accumulator - dataMem[instruct.argument]

I suggest you write two private methods for your GritVM, evaluate and advance. While the machine is RUNNING, the evaluate method contains a switch statement for every instruction in the instruction set and does the proper functionality. The advance method moves the current instruction by the proper amount. This way, you can have something like:

```
while(machineStatus == RUNNING) {
    // Evaluate the current instruction
    long jumpDistance = evaluate(*currentInstruct);

    // Advance the current instruction based on the last evaluation
    advance(jumpDistance);
}
```

It would be a good idea to have a public method that can print out all of the GritVM variables (like above in the example). I suggest something like:

```
void GritVM::printVM(bool printData, bool printInstruction) {
  cout << "****** Output Dump ******" << endl;
  cout << "Status: " <<
           GVMHelper::statusToString(machineStatus) << endl;
  cout << "Accumulator: " << accumulator << endl;
```

```
  if (printData) {
    cout << "*** Data Memory ***" << endl;
    int index = 0;
    vector<long>::iterator it = dataMem.begin();
    while(it != dataMem.end()) {
      long item = (*it);
      cout << "Location " << index++ << ": " << item << endl;
      it++;
    }
  }

  if (printInstruction) {
    cout << "*** Instruction Memory ***" << endl;
    int index = 0;
    list<Instruction>::iterator it = instructMem.begin();
    while(it != instructMem.end()) {
      Instruction item = (*it);
      cout <<   "Instruction " <<
                index++ << ": " <<
                GVMHelper::instructionToString(item.operation) <<
                " " << item.argument << endl;
      it++;
    }
  }
}
```

Try looking through the code of the GVM files and see if you can reverse engineer the functionality.

I would start by getting your load GritVM to load in a GVM file and make sure that all the instruction memory and data memory are working before moving forward. After that, you can work on the run loop that evaluates instructions and moves the current instruction forward.

Extra Credit

There are two opportunities for extra credit on this project.

The first is to write a program in the GVM language that calculates the minimum number of moves for a Towers of Hanoi (TOH) solution. The memory layout of the program after halting should be [N, Result] where N was the number of disks passed into the program and Result is the minimum number of steps. Try to reverse engineering other GVM programs to get your TOH program working. A working program is worth 10 extra points and is defined in the second test case.

The second opportunity is to write your own custom implementation of the list and vector used for the GritVM. You will have to be able to convert to and from the Standard Template Versions of the list and vector when taking in or outputting instruction/data memory. This will also be worth an extra 10 points.

Since these are extra credit opportunities, I will not be answering questions on the implementation details. I will only clarify questions on the requirements of the extra credit.

Rubric

Any code that does not compile will receive a zero for this project.

Criteria	Points
GritVM should be in WAITING state when no program is loaded	2.5
GritVM should throw if the file cannot be loaded and be in state READY if the file is loaded	5
GritVM should run a program if READY and be HALTED when program completes	5
GritVM produces proper output for test.gvm	15
GritVM produces proper output for sumn.gvm	15
GritVM produces proper output for surfarea.gvm	15
GritVM produces proper output for altseq.gvm	15
Student uses List for instruction memory and vector of longs for data memory	12.5
Code uses object oriented design principles (Separate headers and sources, where applicable)	7.5
Code is well documented	7.5
Total Points	**100**

Extra Credit

Criteria	Points
Successfully implemented a TOH GVM program (Test case 2)	10
Student implemented Vector and List classes within the GritVM	10
Total Points	**20**