```
// Sam Borick <sb205@uakron.edu>

#ifndef MACHINE_HPP
#define MACHINE_HPP

#include "test.hpp"
#include "string.hpp"
#include "vector.hpp"
#include "stack.hpp"


// Operation codes. These represent operations that can be executed
// by the virtual machine.
enum
{
  // Basic push/pop
  push_op,  // Push a constant operand
  pop_op,   // Pop an operand
  copy_op,  // Copy the top operand

  // Arithmetic
  add_op,   // Add the top two operands
  sub_op,   // Subtract the top from the lower operands
  mul_op,   // Multiply the top two operands
  div_op,   // Divide the lower from the top
  rem_op,   // Remainder of lower divided by the top

  // Misc.
  print_op, // Pop the top value and print.
  read_op,  // Read a value, push it.
  halt_op,  // Stop executing
};


// Represents an instruction. Every instruction has an operation
// code (one of the values above), and an integer operand.
struct Instruction
{
  Instruction(int o, int a)
    : op(o), arg(a)
  { }

  Instruction(int o)
    : op(o)
  { }

  int op;
  int arg;
};


// Represents the virtual machine. Each machine instance contains
// the source code for a single program.
struct Machine
{
  Machine(std::istream&);

  void run();

  // Program control
  Instruction fetch();

  // Operand stack methods
  int top() const;
  void push(int);
  int pop();

  // Operations
  void copy();
```

```
  void add();
  void sub();
  void mul();
  void div();
  void rem();
  void print();
  void read();
  void halt();

  Vector<Instruction> prog;  // A loaded program
  Stack<int>          stack; // The operand stack

  // Registers
  int pc;
};


#endif
```

```
// Sam Borick <sb205@uakron.edu>

#include "machine.hpp"

#include <map>
#include <iostream>
#include <sstream>


// Returns the op code found in the first n characters of s. Throws an
// exception if the operation name is invalid.
static int
get_op(String const& s)
{
  // A lookup table that maps from strings to opcodes.
  static std::map<String, int> ops {
    {"push", push_op},
    {"pop", pop_op},
    {"copy", copy_op},
    {"add", add_op},
    {"sub", sub_op},
    {"mul", mul_op},
    {"div", div_op},
    {"rem", rem_op},
    {"print", print_op},
    {"read", read_op},
    {"halt", halt_op},
  };

  auto iter = ops.find(s);
  if (iter == ops.end()) {
    String msg = "no such opcode ’" + s + "’";
    throw std::runtime_error(msg);
  }
  return iter->second;
}


int
get_arg(String const& s)
{
  if (s.empty())
    return 0;
  else
    return std::stoi(s);
}


Machine::Machine(std::istream& is)
{
  // Parse instructions from input.
  while (is) {
    String s;
    getline(is, s);
    if (!is)
      break;

    // Search for a ’;’, indicating a comment and strip that from the line.
    std::size_t k = s.find(’;’);
    if (k != String::npos)
      s = s.substr(0, k);

    // Skip empty lines.
    if (s.empty())
      continue;

    // Parse out the opcode and operand.
    std::stringstream ss(s);
    std::string opstr, argstr;
```

```
    ss >> opstr >> argstr;

    int op = get_op(opstr);
    int arg = get_arg(argstr);
    Instruction ins(op, arg);
    prog.push_back(ins);
  }
}


void
Machine::run()
{
  // Start the pc at the first instruction.
  pc = 0;
  int progSize = prog.size(); //This had to be tweeked slightley to fix an issue
 with comparing a signed and unsigned type
  while (pc != progSize) {

    // Get the next instruction.
    Instruction ins = fetch();

    // "Decode" and execute the instruction.
    switch (ins.op) {
      case push_op:
        push(ins.arg);
        break;
      case pop_op:
        pop();
        break;
      case copy_op:
        copy();
        break;
      case add_op:
        add();
        break;
      case sub_op:
        sub();
        break;
      case mul_op:
        mul();
        break;
      case div_op:
        div();
        break;
      case rem_op:
        rem();
        break;
      case print_op:
        print();
        break;
      case read_op:
        read();
        break;
      case halt_op:
        halt();
        break;
    }
  }
}


Instruction
Machine::fetch()
{
  return prog[pc++];
}
```

```
int
Machine::top() const
{
  int output = stack.top();
  return output;
  //throw std::logic_error("not implemented");
}


void
Machine::push(int n)
{
  stack.push(n);
  //throw std::logic_error("not implemented");
}


int
Machine::pop()
{
  int output = stack.top();
  stack.pop();
  return output;

  //throw std::logic_error("not implemented");
}


void
Machine::copy()
{
  stack.push(stack.top());
  //throw std::logic_error("not implemented");
}


void
Machine::add()
{
  int temp1 = pop();
  int temp2 = pop();
  push(temp1 + temp2);
  //throw std::logic_error("not implemented");
}


void
Machine::sub()
{
  int temp1 = pop();
  int temp2 = pop();
  push(temp1 - temp2);
  //row std::logic_error("not implemented");
}


void
Machine::mul()
{
  int temp1 = pop();
  int temp2 = pop();
  push(temp1 * temp2);
  //throw std::logic_error("not implemented");
}


void
Machine::div()
{
```

```
  int temp1 = pop();
  int temp2 = pop();
  push(temp2/temp1);
  //throw std::logic_error("not implemented");
}


void
Machine::rem()
{
  int temp1 = pop();
  int temp2 = pop();
  push(temp2%temp1);
  //throw std::logic_error("not implemented");
}


void
Machine::print()
{
  std::cout << pop();
  //throw std::logic_error("not implemented");
}


void
Machine::read()
{
  int input;
  std::cin >> input;
  push(input);
  //throw std::logic_error("not implemented");
}


void
Machine::halt()
{
  pc = prog.size();
}
```

```
// Sam Borick <sb205@uakron.edu>

#ifndef STACK_HPP
#define STACK_HPP

#include "test.hpp"
#include "vector.cpp"
#include <stack>



template<typename T>
//using Stack = std::stack<T>;

struct Stack{
  Vector<T> vec;

  Stack(){}//vec is already initialized to empty

  Stack(const Stack & S){
    vec = S->vec;
  }

  Stack& operator=(const Stack & S){
    Stack p = S;
    swap(*this, p);
    return *this;
  }

  void swap(Stack & a, Stack & b){
    swap(a->vec, b->vec);
  }

  bool empty(){
    return (vec.size() == 0);
  }

  size_t size(){
    return vec.size();
  }

  const T top()const{
    return vec.back();
  }

 /* T & top(){
    return vec.back();
  }*/

  void push(T input){
    vec.push_back(input);
  }

  void pop(){
    assert(!vec.empty());
    vec.pop_back();
  }


};



#endif
```

```
// Sam Borick <sb205@uakron.edu>

#ifndef Vector_HPP
#define Vector_HPP

#include "test.hpp"
#include "memory.hpp"
#include <initializer_list>

template<typename T>
struct Vector
{
  Vector(std::initializer_list<T> list)
  :base(), last(), limit()
{
  reserve(list.size());
  for (T const& s : list)
    push_back(s);
}

  T* base = nullptr;
  T* last = nullptr;
  T* limit = nullptr;

  Vector(){
   // reserve(8);
  }

  Vector(const Vector& v){
    reserve(v.size());
    base = last;
    last = uninitialized_copy(v.base, v.last, base);
}

  Vector& operator=(const Vector & v){  //this is a neat optimization I found on
 stackoverflow.  I think it's
  //really elegant and now I understand the difference between copy construction
 and copy assingnment better
    Vector p = v;
    swap(*this, p);
    return *this;
  }

  T& operator[](const size_t pos)const{
    assert(pos >=0);
    assert(pos < size());
    return base[pos];
  }

   ~Vector(){
    initialized_destroy(base, last);
    deallocate(base);
  }

  void clear(){
    resize(0);
  }

  size_t size()const{
    return last - base;
  }

  void swap(Vector & v1, Vector & v2){
    std::swap(v1.base, v2.base);
    std::swap(v1.last, v2.last);
    std::swap(v1.limit, v2.limit);
  }

  void reserve(std::size_t n){
    if(!base){
      base = allocate<T>(n);
      last = base;
      limit = n + base;
    }else if(n <= capacity()){
    }else{
      T* p = allocate<T>(n);
      T* q = p;
      for(T*i = base; i != last; ++i){
        new(q)T(*i);
        ++q;
      }
      for(T*i = base; (i==last); ++i){
        i ->~T();
      }
      deallocate<T>(base);
      base = p;
      last = q;
      limit = base + n;
    }
  }

  void resize(std::size_t n){
    if(n == size()){
    }else if(n < size()){
      //int counter = size() - n;
      for(int counter= size() - n; counter > 0; --counter){
         destroy(--last);
      }
    }else{
      //int counter = n - size();
      for(int counter= n - size(); counter >= 0; --counter){
        push_back("");//yeah, gross
        //TODO: make this better with construct
      }
    }
  }

  bool empty()const{
    return (base == last);
  }

  void push_back(T const & s){
    if(!base){
      reserve(8);
    }else if(last == limit){
      reserve(2*capacity());
    }
    construct(last++, s);
  }

  void pop_back(){
    assert(!empty());
    destroy(--last);
  }

  size_t capacity()const{
    return limit - base;
  }

  const T& back()const{
    return *(last-1);
  }

  T const* data(){
    return base;
  }

  using iterator = T*;
```

```
  using const_iterator = T*;

  iterator begin(){
    return base;
  }

  iterator end(){
    return last;
  }

  const_iterator begin()const{
    return base;
  }

  const_iterator end()const{
    return last;
  }

};

template<typename T>
bool operator==(Vector<T> const &, Vector<T> const &);
template<typename T>
bool operator!=(Vector<T> const &, Vector<T> const &);
template<typename T>
bool operator<(Vector<T> const &, Vector<T> const &);
template<typename T>
bool operator>(Vector<T> const &, Vector<T> const &);
template<typename T>
bool operator<=(Vector<T> const &, Vector<T> const &);
template<typename T>
bool operator>=(Vector<T> const &, Vector<T> const &);

template<typename T>
bool operator ==(Vector<T> const & v1, Vector<T> const & v2){
 /* std::size_t counter = 0;
  if(v1.size() != v2.size()){
    return false;
  }
  while (counter < v1.size()) {
    if(v1.base+counter != v2.base+counter){
      return false;
    }
    ++counter;
  }
  return true;
  */
  return std::equal(v1.base, v1.last, v2.base);
}
template<typename T>
bool operator !=(Vector<T> const &v1, Vector<T> const & v2){
  return !(v1==v2);
}
template<typename T>
bool operator<(Vector<T> const &v1, Vector<T> const & v2){
  return std::lexicographical_compare(v1.base, v1.last, v2.base, v2.last);
}
template<typename T>
bool operator>(Vector<T> const& v1, Vector<T> const & v2){
  return std::lexicographical_compare(v2.base, v2.last, v1.base, v2.last);
}
template<typename T>
bool operator<=(Vector<T> const& v1, Vector<T> const & v2){
  return !(v1>v2);
}
template<typename T>
bool operator>=(Vector<T> const& v1, Vector<T> const & v2){
  return!(v1<v2);
}
```

```
#endif

/*#ifndef VECTOR_HPP  //old vector re-direct
#define VECTOR_HPP

#include "test.hpp"

#include <vector>


template<typename T>
using Vector = std::vector<T>;


#endif*/
```

```
// $NAME <$ID@uakron.edu>

#ifndef STRING_HPP
#define STRING_HPP

#include "test.hpp"

#include <string>


using String = std::string;

#endif
```