```cpp
// Karthik Venkat <kv39@zips.uakron.edu>

#ifndef MACHINE_HPP
#define MACHINE_HPP

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

enum
{
  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
};

// 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
```



```cpp
  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
```

```
//Karthik Venkat <kv39@zips.uakron.edu>

#include "machine.hpp"

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

// 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.data());
  }
  return iter->second;
}

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

Machine::Machine(std::istream& is)
{
  // Parse instructions from input
  typedef std::basic_string<char> String;

  while (is)
  {
    String s;
    std::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.data());
    std::string opstr, argstr;
    ss >> opstr >> argstr;

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


void Machine::run()
{
  // Start the pc at the first instruction.
  pc = 0;

  while (pc != prog.size())
  {

    // 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() //fetch instruction
{
  return prog[pc++];
}

int Machine::top() const //top element
{
  return stack.top();
}

void Machine::push(int n) //push to top
{
  stack.push(n);
}

int Machine::pop() //pop from top
{
  int returnthis = stack.top();
  stack.pop();
  return returnthis;
}

void Machine::copy() //push a copy of the top operand on the stack

{
  stack.push(stack.top());
}

void Machine::add() //Add top two elements
{
  int result = pop();
  result = result + pop();
  stack.push(result);
}

void Machine::sub() //Subtract top two elements
{
  int result = stack.top();
  stack.pop();
  result = result - stack.top();
  stack.pop();
  stack.push(result);
}

void Machine::mul() //multiply top two elements
{
  int result = stack.top();
  stack.pop();
  result = result * stack.top();
  stack.pop();
  stack.push(result);
}

void Machine::div() //divide top two elements
{
```



```
  int result = stack.top();
  stack.pop();
  assert(result != 0);
  result = stack.top() / result;
  stack.pop();
  stack.push(result);
}

void Machine::rem() //return mod of top 2 elements
{
  int result = stack.top();
  stack.pop();
  assert(result != 0);
  result = stack.top() % result;
  stack.pop();
  stack.push(result);
}

void Machine::print() //print top element
{
  int output = stack.top();
  stack.pop();
  std::cout << output << std::endl;
}

void Machine::read() //read input
{
  int input;
  std::cin >> input;
  stack.push(input);
}

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

```cpp
// Karthik Venkat <kv39@zips.uakron.edu>
//Stack implementation done with the assistance of Adam J Browne

#ifndef STACK_HPP
#define STACK_HPP

#include "test.hpp"
#include "vector.hpp"

template<typename T> struct Stack
{
  Vector<T> member;
  Stack(); //default constructor
  Stack(Stack<T> const&); //copy constructor
  int top() const;
  int& top();
  bool empty() const;
  std::size_t size() const;
  void push(T const&); //push to top
  void pop(); //pop off top
  Stack<T>& operator=(Stack<T> const&); //copy assign
};

template<typename T> Stack<T>::Stack() //default construction
  :member() {}

template<typename T> Stack<T>::Stack(Stack<T> const& v) //copy construction
  :member(v.member) {}

template<typename T> int Stack<T>::top() const //return top element
{
  return member.back();
}

template<typename T> int& Stack<T>::top() //return top of stack
{
   return member.back();
}

template<typename T> bool Stack<T>::empty() const //check for empty stack
{
  return member.empty();
}

template<typename T> size_t Stack<T>::size() const //Return size of stack
{
  return member.size();
}

template<typename T> void Stack<T>::push(T const& n) //push to top
{
  member.push_back(n);
}

template<typename T> void Stack<T>::pop() //pop off top
{
  member.pop_back();
}

template<typename T> Stack<T>& Stack<T>::operator=(Stack<T> const& v) //assign
{
  member = v.member;
```



```cpp
  return *this;
}

#endif
```

```
// Karthik Venkat <kv39@zips.uakron.edu>
//this stack, and those are (c) Andrew Sutton 2016

#ifndef Vector_HPP
#define Vector_HPP

#include "memory.hpp"
#include "test.hpp"
#include <algorithm>
#include <initializer_list>
template <typename T>
struct Vector
{
  T *base, *last, *limit;

  using iterator = T*;
  using const_iterator = T const*;

  iterator begin()
  {
    return base;
  }

  iterator end()
  {
     return last;
  }

  const_iterator begin() const
  {
     return base;
  }

  const_iterator end() const
  {
     return last;
  }

  //default constructor
  Vector();
  //initializer_list constructor.
  Vector(std::initializer_list<T>);
  //Copy constructor
  Vector(Vector const&);
  //destructor, release memory.
  ~Vector();

  T& operator[](std::size_t n)
  {
    assert(n < size()); return base[n];
  }

  T operator[](std::size_t n) const
  {
     assert(n < size()); return base[n];
  }

  T* data() const
  {
     return base;
  }
```



```
  void reserve(std::size_t n);
  bool empty() const; //empty if last == base
  std::size_t size() const; //size = last-base
  std::size_t capacity() const; //capacity = limit-base
  void push_back(T const&);
  void pop_back();
  void resize(size_t n); //resize the vector to a size of size n.
  void clear(); //makes vector empty but does not release spare capacity.
  T back() const;
  T& back();
  Vector& operator=(Vector const&);
};

template<typename T>
Vector<T>::Vector() //default construction
  :base(), last(), limit()
{ }

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

template<typename T>
Vector<T>::Vector(Vector<T> const& v)
  :base(allocate<T>(v.size())),
  last(uninitialized_copy(v.base, v.limit, base)),
  limit(base + v.size())
{ }

template<typename T>
Vector<T>::~Vector()
{
  //destructor
  initialized_destroy(base, last);
  deallocate(base);
}

template<typename T>
bool Vector<T>::empty() const
{
   if (last == base) return true;
   return false;
 }

template<typename T>
std::size_t Vector<T>::size() const
{
   return last-base;
}
template<typename T>
std::size_t Vector<T>::capacity() const
 {
    return limit-base;
}

template<typename T>
```

```
T Vector<T>::back() const
{
   return *(last-1);
}

template<typename T>
T& Vector<T>::back()
{
   return *(last-1);
 }

template<typename T>
void Vector<T>::reserve(std::size_t n)
{
  if(n > capacity())
  {
    if(!base)
     {
      base = allocate<T>(n);
      last = base;
      limit = base + n;
    }
    else
    {
      //allocate new memory of type T for size n
      T *p = allocate<T>(n); //new base
      T *q = p; //new last
      T *i = uninitialized_copy(base, last, q); //copy new storage
      destroy(i);
      deallocate(base);
      limit = p + n; //update
      base = p;
      last = q;
    }
  }
}

template<typename T>
void Vector<T>::resize(size_t n) //resize the vector.
{
  if(n > size())
    while(size()!=n) push_back(T()); //pushback until specified size.
  else
    while(size()!=n) pop_back(); //popback until vector is specified size.
}

template<typename T>
void Vector<T>::push_back(T const& s)
{
  if(!base)
    reserve(8); //not too big, not too small.
  else if(limit==last)
    reserve(2*capacity());
  construct(last++, s); //inplace construction
}

template<typename T>
void Vector<T>::pop_back()
{
  assert(!empty());
  destroy(--last);
}
```



```

template<typename T>
void Vector<T>::clear()
{
  initialized_destroy(base, last);
  last = base;
}

template<typename T>
Vector<T>& Vector<T>::operator=(Vector<T> const& s)
{
  if(this == &s) //self-assignment guard.
    return *this;
  clear(); //cleanup
  deallocate(base);
  base = allocate<T>(s.size()); //allocate memory of size of the object
  last = uninitialized_copy(s.begin(), s.limit, begin());
  limit = base + s.size();
  return *this;
}

template<typename T>
bool operator==(Vector<T> const& a, Vector<T> const& b)
{
   return (a.size() == b.size()) && std::equal(a.begin(), a.end(), b.begin(
}
template<typename T>
bool operator!=(Vector<T> const& a, Vector<T> const& b)
{
   return !(a == b);
}
template<typename T>
bool operator<(Vector<T> const& a, Vector<T> const& b)
{
   return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end
}
template<typename T>
bool operator>(Vector<T> const& a, Vector<T> const& b)
{
   return !(a < b);
}
template<typename T>
bool operator<=(Vector<T> const& a, Vector<T> const& b)
{
  if (a == b || std::lexicographical_compare(a.begin(), a.end(),
 b.begin(), b.end())) //Split for neatness
   return true;
  return false;
 }

template<typename T>
bool operator>=(Vector<T> const& a, Vector<T> const& b)
{
  if (a == b || !std::lexicographical_compare(a.begin(), a.end(),
 b.begin(), b.end())) //Split for neatness
   return true;
 return false;
}

#endif
```

```
// Karthik Venkat <kv39@zips.uakron.edu>

#ifndef STRING_HPP
#define STRING_HPP
#include "test.hpp"
#include "test.hpp"

#include <cstring>
#include <iosfwd>


struct String
{
  private:
  std::size_t len;
  char *str;

  public:

  static constexpr std::size_t npos = -1;
  String(); //Default constructor
  String(const char* s) ; //Constructor for string with value
  String(const String &s); //copy constructor
  String(char const *c, std::size_t); //Constructor for bounded strings
  String(std::nullptr_t)
  {
    assert(0);
  }///When nullptr is passed to the string

  ~String(); //Destructor

  char *data() const //Return the string contents
  {
    return str;
  }

  std::size_t size() const //Return length of the string
  {
    return len;
  }

  bool empty() const //Check for empty string
  {
    return (len == 0);
  }

  std::size_t find(int ch) const; //For find operation

  String substr(std::size_t, std::size_t) const; //To find substring in string

  char &operator [] (std::size_t a) //For character subscript access
  {
    assert (a < len && a >= 0);
    return str[a];
  }

  char operator [] (std::size_t a) const //For character subscript access
  {
    assert (a < len && a >= 0);
    return str[a];
  }
```



```
  String &operator = (String const &s) //Assignment operator
  {
      if(this!= &s)
      {
        delete []str;
        len = s.len;
        str = new char[len + 1];
        strcpy(str, s.str);
      }
      return *this;
  }

  String &operator += (String const &s) //Copy assign operator
  {
    char *p = new char [(len + s.len) +1];
    strcpy (p, str);
    strcpy (p + len, s.str);
    std::swap(str, p);
    len = len + s.len;
    delete [] p;
    return *this;
  }

};

//Overload for concatenation
String operator + (const String &s1, const String &s2);

//Overloads for equality and inequality
bool operator == (const String& s1, const String& s2);
bool operator == (const String s1, char const *c);
bool operator == (char const *c, const String s2);
bool operator != (const String s1, const String s2);
bool operator != (const String s1, char const *c);
bool operator != (char const *c, const String s2);
//Overloads for greater than, less than and/or equal to operators
bool operator <= (const String s1, const String s2);
bool operator <= (const String s1, char const *c);
bool operator <= (char const *c, const String s2);
bool operator >= (const String s1, const String s2);
bool operator >= (const String s1, char const *c);
bool operator >= (char const *c, const String s2);
bool operator < (const String s1, const String s2);
bool operator < (const String s1, char const *c);
bool operator < (char const *c, const String s2);
bool operator > (const String s1, const String s2);
bool operator > (const String s1, char const *c);
bool operator > (char const *c, const String s2);

// Output stream overload
std::ostream &operator << (std::ostream &os, String const &str);

#endif
```