HOMEWORK 3
==========
Problem 1:

Since all Creatures have a name, and since the way you find out the name is going to be the same for all kinds of creatures, the name should be a data member of Creature, and the function to retrieve it need not be virtual.

The different kinds of creatures may have different ways of moving, and may or may not be mortal, so these functions should be virtual. Since most creatures are mortal, it's convenient to have Creature::mortal have an implementation (that returns true) that derived classes may inherit if they wish. There is no reasonable default behavior for move, so this should be pure virtual.

Observe how the constructors for the derived classes pass the name to the Creature constructor. Examine Giant's constructor especially.

```
// ====================================== Creature
class Creature
{
  public:
    Creature(string nm);
    string name() const;
    virtual string move() const = 0;
    virtual bool mortal() const;
    virtual ~Creature() {}
  private:
    string m_name;
};

Creature::Creature(string nm)
 : m_name(nm)
{}

string Creature::name() const
{
  return m_name;
}

bool Creature::mortal() const
{
  return true;
}

// ====================================== Phoenix
class Phoenix : public Creature
{
  public:
```

```
    Phoenix(string nm);
    virtual string move() const;
    virtual bool mortal() const;
    virtual ~Phoenix();
};

Phoenix::Phoenix(string nm)
 : Creature(nm)
{}

string Phoenix::move() const
{
  return "fly";
}

bool Phoenix::mortal() const
{
  return false;
}

Phoenix::~Phoenix()
{
  cout << "Destroying " << name() << " the phoenix." << endl;
}

// ====================================== Giant
class Giant : public Creature
{
  public:
    Giant(string nm, int wt);
    virtual string move() const;
    virtual ~Giant();
  private:
    int m_weight;
};

Giant::Giant(string nm, int wt)
 : Creature(nm), m_weight(wt)
{}

string Giant::move() const
{
  if (m_weight < 20)
    return "tromp";
  else
    return "lumber";
}
```

```
Giant::~Giant()
{
  cout << "Destroying " << name() << " the giant." << endl;
}

// ====================================== Centaur
class Centaur : public Creature
{
  public:
    Centaur(string nm);
    virtual string move() const;
    virtual ~Centaur();
  private:
};

Centaur::Centaur(string nm)
 : Creature(nm)
{}

string Centaur::move() const
{
  return "trot";
}

Centaur::~Centaur()
{
  cout << "Destroying " << name() << " the centaur." << endl;
}
```

Problem 2:

```
  // Return false if the somePredicate function returns false for at
  // least one of the array elements; return true otherwise.
bool allTrue(const string a[], int n)
{
    if (n <= 0)
        return true;
    if ( ! somePredicate(a[0]))
        return false;
    return allTrue(a+1, n-1);
}

  // Return the number of elements in the array for which the
  // somePredicate function returns false.
int countFalse(const string a[], int n)
{
    if (n <= 0)
        return 0;
    int f = (somePredicate(a[0]) ? 0 : 1);
```

```
    return f + countFalse(a+1, n-1);
}

  // Return the subscript of the first element in the array for which
  // the somePredicate function returns false.  If there is no such
  // element, return -1.
int firstFalse(const string a[], int n)
{
    if (n <= 0)
        return -1;
    if ( ! somePredicate(a[0]))
        return 0;
    int k = firstFalse(a+1, n-1);
    if (k == -1)
        return -1;
    return 1 + k;  // element k of "the rest of a" is element 1+k of a
}

  // Return the subscript of the least string in the array (i.e.,
  // the smallest subscript m such that a[m] <= a[k] for all
  // k from 0 to n-1).  If the array has no elements to examine,
  // return -1.
int indexOfLeast(const string a[], int n)
{
    if (n <= 0)
        return -1;
    if (n == 1)
        return 0;
    int k = 1 + indexOfLeast(a+1, n-1);  // indexOfLeast can't return -1 here
    return a[0] <= a[k] ? 0 : k;

    // Here's an alternative for the last two lines above:
    // int k = indexOfLeast(a, n-1);  // indexOfLeast can't return -1 here
    // return a[k] <= a[n-1] ? k : n-1;
}

  // If all n2 elements of a2 appear in the n1 element array a1, in
  // the same order (though not necessarily consecutively), then
  // return true; otherwise (i.e., if the array a1 does not include
  // a2 as a not-necessarily-contiguous subsequence), return false.
  // (Of course, if a2 is empty (i.e., n2 is 0), return true.)
  // For example, if a1 is the 7 element array
  //    "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"
  // then the function should return true if a2 is
  //    "kyle" "kenny" "butters"
  // or
  //    "kyle" "cartman" "cartman"
  // and it should return false if a2 is
```

```
  //    "kyle" "butters" "kenny"
  // or
  //    "stan" "kenny" "kenny"
bool includes(const string a1[], int n1, const string a2[], int n2)
{
    if (n2 <= 0)
        return true;
    if (n1 < n2)
        return false;

      // If we get here, a1 and a2 are nonempty
    if (a1[0] == a2[0])
        return includes(a1+1, n1-1, a2+1, n2-1);  // rest of a1, rest of a2
    else
        return includes(a1+1, n1-1, a2, n2);  // rest of a1, all of a2
}
Problem 3:

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    if (sr == er  &&  sc == ec)
        return true;

    maze[sr][sc] = '@';  // anything non-'.' will do

    if (maze[sr-1][sc] == '.'  &&  pathExists(maze, sr-1, sc, er, ec))
        return true;
    if (maze[sr+1][sc] == '.'  &&  pathExists(maze, sr+1, sc, er, ec))
        return true;
    if (maze[sr][sc-1] == '.'  &&  pathExists(maze, sr, sc-1, er, ec))
        return true;
    if (maze[sr][sc+1] == '.'  &&  pathExists(maze, sr, sc+1, er, ec))
        return true;

    return false;
}
or

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    if (maze[sr][sc] != '.')
        return false;

    if (sr == er  &&  sc == ec)
        return true;

    maze[sr][sc] = '@';  // anything non-'.' will do
```

```
    if (pathExists(maze, sr-1, sc, er, ec))
        return true;
    if (pathExists(maze, sr+1, sc, er, ec))
        return true;
    if (pathExists(maze, sr, sc-1, er, ec))
        return true;
    if (pathExists(maze, sr, sc+1, er, ec))
        return true;

    return false;
}
```

Problem 4:

```
  // Return the number of ways that all n2 elements of a2 appear
  // in the n1 element array a1 in the same order (though not
  // necessarily consecutively).  The empty sequence appears in a
  // sequence of length n1 in 1 way, even if n1 is 0.
  // For example, if a1 is the 7 element array
  //    "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"
  // then for this value of a2            the function must return
  //    "stan" "kenny" "cartman"                     1
  //    "stan" "cartman" "butters"                   2
  //    "kenny" "stan" "cartman"                     0
  //    "kyle" "cartman" "butters"                   3
int countIncludes(const string a1[], int n1, const string a2[], int n2)
{
    if (n2 <= 0)
        return 1;
    if (n1 < n2)
        return 0;

      // If we get here, a1 and a2 are nonempty
    int t = countIncludes(a1+1, n1-1, a2, n2);  // rest of a1, all of a2
    if (a1[0] == a2[0])
        t += countIncludes(a1+1, n1-1, a2+1, n2-1);  // rest of a1, rest of a2
    return t;
}

  // Exchange two strings
void exchange(string& x, string& y)
{
    string t = x;
    x = y;
    y = t;
}

  // Rearrange the elements of the array so that all the elements
  // whose value is < divider come before all the other elements,
```

```
  // and all the elements whose value is > divider come after all
  // the other elements.  Upon return, firstNotLess is set to the
  // index of the first element in the rearranged array that is
  // >= divider, or n if there is no such element, and firstGreater is
  // set to the index of the first element that is > divider, or n
  // if there is no such element.
  // In other words, upon return from the function, the array is a
  // permutation of its original value such that
  //   * for 0 <= i < firstNotLess, a[i] < divider
  //   * for firstNotLess <= i < firstGreater, a[i] == divider
  //   * for firstGreater <= i < n, a[i] > divider
  // All the elements < divider end up in no particular order.
  // All the elements > divider end up in no particular order.
void divide(string a[], int n, string divider,
                                    int& firstNotLess, int& firstGreater)
{
    if (n < 0)
        n = 0;

      // It will always be the case that just before evaluating the loop
      // condition:
      //  firstNotLess <= firstUnknown and firstUnknown <= firstGreater
      //  Every element earlier than position firstNotLess is < divider
      //  Every element from position firstNotLess to firstUnknown-1 is
      //    == divider
      //  Every element from firstUnknown to firstGreater-1 is not known yet
      //  Every element at position firstGreater or later is > divider

    firstNotLess = 0;
    firstGreater = n;
    int firstUnknown = 0;
    while (firstUnknown < firstGreater)
    {
        if (a[firstUnknown] > divider)
        {
            firstGreater--;
            exchange(a[firstUnknown], a[firstGreater]);
        }
        else
        {
            if (a[firstUnknown] < divider)
            {
                exchange(a[firstNotLess], a[firstUnknown]);
                firstNotLess++;
            }
            firstUnknown++;
        }
    }
```

```
}

  // Rearrange the elements of the array so that
  // a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]
  // If n <= 1, do nothing.
void order(string a[], int n)
{
    if (n <= 1)
        return;

      // Divide using a[0] as the divider (any element would do).
    int firstNotLess;
    int firstGreater;
    divide(a, n, a[0], firstNotLess, firstGreater);

      // sort the elements < divider
    order(a, firstNotLess);

      // sort the elements > divider
    order(a+firstGreater, n-firstGreater);
}
```

==========

Project 1

==========

```
// Arena.h

#ifndef ARENA_INCLUDED
#define ARENA_INCLUDED

#include "globals.h"
#include "History.h"

class Player;
class Robot;

class Arena
{
  public:
        // Constructor/destructor
    Arena(int nRows, int nCols);
    ~Arena();

        // Accessors
    int      rows() const;
    int      cols() const;
    Player*  player() const;
```

```
    int      robotCount() const;
    int      nRobotsAt(int r, int c) const;
    bool     determineNewPosition(int& r, int& c, int dir) const;
    void     display() const;
    History& history();

        // Mutators
    bool   addRobot(int r, int c);
    bool   addPlayer(int r, int c);
    bool   attackRobotAt(int r, int c, int dir);
    bool   moveRobots();

  private:
    int     m_rows;
    int     m_cols;
    Player* m_player;
    Robot*  m_robots[MAXROBOTS];
    int     m_nRobots;
    History m_history;
};

#endif // ARENA_INCLUDED

// Arena.cpp

#include "Arena.h"
#include "Player.h"
#include "Robot.h"
#include "History.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Arena::Arena(int nRows, int nCols)
 : m_rows(nRows), m_cols(nCols), m_player(nullptr), m_nRobots(0),
   m_history(nRows, nCols)
{
    if (nRows <= 0  ||  nCols <= 0  ||  nRows > MAXROWS  ||  nCols > MAXCOLS)
    {
        cout << "***** Arena created with invalid size " << nRows << " by "
             << nCols << "!" << endl;
        exit(1);
    }
}

Arena::~Arena()
{
```

```
    for (int k = 0; k < m_nRobots; k++)
        delete m_robots[k];
    delete m_player;
}

int Arena::rows() const
{
    return m_rows;
}

int Arena::cols() const
{
    return m_cols;
}

Player* Arena::player() const
{
    return m_player;
}

int Arena::robotCount() const
{
    return m_nRobots;
}

int Arena::nRobotsAt(int r, int c) const
{
    int count = 0;
    for (int k = 0; k < m_nRobots; k++)
    {
        const Robot* rp = m_robots[k];
        if (rp->row() == r  &&  rp->col() == c)
            count++;
    }
    return count;
}

bool Arena::determineNewPosition(int& r, int& c, int dir) const
{
    switch (dir)
    {
      case UP:     if (r <= 1)      return false; else r--; break;
      case DOWN:   if (r >= rows()) return false; else r++; break;
      case LEFT:   if (c <= 1)      return false; else c--; break;
      case RIGHT:  if (c >= cols()) return false; else c++; break;
      default:     return false;
    }
    return true;
```

```
}

void Arena::display() const
{
      // Position (row,col) in the arena coordinate system is represented in
      // the array element grid[row-1][col-1]
    char grid[MAXROWS][MAXCOLS];
    int r, c;

        // Fill the grid with dots
    for (r = 0; r < rows(); r++)
        for (c = 0; c < cols(); c++)
            grid[r][c] = '.';

        // Indicate each robot's position
    for (int k = 0; k < m_nRobots; k++)
    {
        const Robot* rp = m_robots[k];
        char& gridChar = grid[rp->row()-1][rp->col()-1];
        switch (gridChar)
        {
          case '.':  gridChar = 'R'; break;
          case 'R':  gridChar = '2'; break;
          case '9':  break;
          default:   gridChar++; break;  // '2' through '8'
        }
    }

        // Indicate player's position
    if (m_player != nullptr)
    {
          // Set the char to '@', unless there's also a robot there,
          // in which case set it to '*'.
        char& gridChar = grid[m_player->row()-1][m_player->col()-1];
        if (gridChar == '.')
            gridChar = '@';
        else
            gridChar = '*';
    }

        // Draw the grid
    clearScreen();
    for (r = 0; r < rows(); r++)
    {
        for (c = 0; c < cols(); c++)
            cout << grid[r][c];
        cout << endl;
    }
```

```
    cout << endl;

        // Write message, robot, and player info
    cout << endl;
    cout << "There are " << robotCount() << " robots remaining." << endl;
    if (m_player == nullptr)
        cout << "There is no player." << endl;
    else
    {
        if (m_player->age() > 0)
            cout << "The player has lasted " << m_player->age() << " steps." <<
endl;
        if (m_player->isDead())
            cout << "The player is dead." << endl;
    }
}

History& Arena::history()
{
    return m_history;
}

bool Arena::addRobot(int r, int c)
{
      // Dynamically allocate a new Robot and add it to the arena
    if (m_nRobots == MAXROBOTS)
        return false;
    m_robots[m_nRobots] = new Robot(this, r, c);
    m_nRobots++;
    return true;
}

bool Arena::addPlayer(int r, int c)
{
      // Don't add a player if one already exists
    if (m_player != nullptr)
        return false;

      // Dynamically allocate a new Player and add it to the arena
    m_player = new Player(this, r, c);
    return true;
}

bool Arena::attackRobotAt(int r, int c, int dir)
{
      // Attack one robot.  Returns true if a robot was attacked and destroyed,
      // false otherwise (no robot there, or the attack did not destroy the
      // robot).
```

```
    int k = 0;
    for ( ; k < m_nRobots; k++)
    {
        if (m_robots[k]->row() == r  &&  m_robots[k]->col() == c)
            break;
    }
    if (k < m_nRobots  &&  m_robots[k]->getAttacked(dir))  // robot dies
    {
        m_history.record(m_player->row(), m_player->col());
        delete m_robots[k];
        m_robots[k] = m_robots[m_nRobots-1];
        m_nRobots--;
        return true;
    }
    return false;
}

bool Arena::moveRobots()
{
    for (int k = 0; k < m_nRobots; k++)
    {
        Robot* rp = m_robots[k];
        rp->move();
        if (rp->row() == m_player->row()  &&  rp->col() == m_player->col())
            m_player->setDead();
    }

      // return true if the player is still alive, false otherwise
    return ! m_player->isDead();
}

// Game.h

#ifndef GAME_INCLUDED
#define GAME_INCLUDED

class Arena;

class Game
{
  public:
        // Constructor/destructor
    Game(int rows, int cols, int nRobots);
    ~Game();

        // Mutators
    void play();
```

```
  private:
    Arena* m_arena;
};

#endif // GAME_INCLUDED

// Game.cpp

#include "Game.h"
#include "Arena.h"
#include "Player.h"
#include "History.h"
#include "globals.h"
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

Game::Game(int rows, int cols, int nRobots)
{
    if (nRobots < 0)
    {
        cout << "***** Cannot create Game with negative number of robots!" << endl;
        exit(1);
    }
    if (nRobots > MAXROBOTS)
    {
        cout << "***** Trying to create Game with " << nRobots
             << " robots; only " << MAXROBOTS << " are allowed!" << endl;
        exit(1);
    }
    if (rows == 1  &&  cols == 1  &&  nRobots > 0)
    {
        cout << "***** Cannot create Game with nowhere to place the robots!" <<
endl;
        exit(1);
    }

        // Create arena
    m_arena = new Arena(rows, cols);

        // Add player
    int rPlayer = randInt(1, rows);
    int cPlayer = randInt(1, cols);
    m_arena->addPlayer(rPlayer, cPlayer);

      // Populate with robots
    while (nRobots > 0)
```

```
    {
        int r = randInt(1, rows);
        int c = randInt(1, cols);
          // Don't put a robot where the player is
        if (r == rPlayer  &&  c == cPlayer)
            continue;
        m_arena->addRobot(r, c);
        nRobots--;
    }
}

Game::~Game()
{
    delete m_arena;
}

void Game::play()
{
    Player* p = m_arena->player();
    if (p == nullptr)
    {
        m_arena->display();
        return;
    }
    do
    {
        m_arena->display();
        cout << endl;
        cout << "Move (u/d/l/r//h/q): ";
        string action;
        getline(cin,action);
        if (action.size() == 0)  // player stands
            p->stand();
        else
        {
            switch (action[0])
            {
              default:   // if bad move, nobody moves
                cout << '\a' << endl;  // beep
                continue;
              case 'q':
                return;
              case 'u':
              case 'd':
              case 'l':
              case 'r':
                p->moveOrAttack(decodeDirection(action[0]));
                break;
```

```
            case 'h':
              m_arena->history().display();
              cout << "Press enter to continue.";
              cin.ignore(10000, '\n');
              continue;  // doesn't count as a turn, so don't move robots
           }
        }
        m_arena->moveRobots();
    } while ( ! m_arena->player()->isDead()  &&  m_arena->robotCount() > 0);
    m_arena->display();
}

// History.h

#ifndef HISTORY_INCLUDED
#define HISTORY_INCLUDED

#include "globals.h"

class History
{
  public:
    History(int nRows, int nCols);

        // Accessors
    void display() const;

        // Mutators
    bool record(int r, int c);

  private:
    int  m_grid[MAXROWS][MAXCOLS];
    int  m_rows;
    int  m_cols;
};

#endif // HISTORY_INCLUDED

// History.cpp

#include "History.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;

History::History(int nRows, int nCols)
 : m_rows(nRows), m_cols(nCols)
```

```
{
    if (nRows <= 0  ||  nCols <= 0  ||  nRows > MAXROWS  ||  nCols > MAXCOLS)
    {
        cout << "***** History created with invalid size " << nRows << " by "
             << nCols << "!" << endl;
        exit(1);
    }
    for (int r = 0; r < m_rows; r++)
       for (int c = 0; c < m_cols; c++)
            m_grid[r][c] = 0;
}

void History::display() const
{
    clearScreen();

      // Draw the grid
    for (int r = 0; r < m_rows; r++)
    {
        for (int c = 0; c < m_cols; c++)
        {
            char ch = '.';
            int n = m_grid[r][c];
            if (n >= 26)
                ch = 'Z';
            else if (n > 0)
                ch = 'A' + n-1;
            cout << ch;
        }
        cout << endl;
    }
    cout << endl;
}

bool History::record(int r, int c)
{
    if (r <= 0  ||  c <= 0  ||  r > m_rows  ||  c > m_cols)
        return false;
    m_grid[r-1][c-1]++;
    return true;
}

// Player.h

#ifndef PLAYER_INCLUDED
#define PLAYER_INCLUDED

class Arena;
```

```
class Player
{
  public:
        // Constructor
    Player(Arena *ap, int r, int c);

        // Accessors
    int  row() const;
    int  col() const;
    int  age() const;
    bool isDead() const;

        // Mutators
    void   stand();
    void   moveOrAttack(int dir);
    void   setDead();

  private:
    Arena* m_arena;
    int    m_row;
    int    m_col;
    int    m_age;
    bool   m_dead;
};

#endif // PLAYER_INCLUDED

// Player.cpp

#include "Player.h"
#include "Arena.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Player::Player(Arena* ap, int r, int c)
 : m_arena(ap), m_row(r), m_col(c), m_age(0), m_dead(false)
{
    if (ap == nullptr)
    {
        cout << "***** The player must be in some Arena!" << endl;
        exit(1);
    }
    if (r < 1  ||  r > ap->rows()  ||  c < 1  ||  c > ap->cols())
    {
        cout << "**** Player created with invalid coordinates (" << r
             << "," << c << ")!" << endl;
```

```
        exit(1);
    }
}

int Player::row() const
{
    return m_row;
}

int Player::col() const
{
    return m_col;
}

int Player::age() const
{
    return m_age;
}

void Player::stand()
{
    m_age++;
}

void Player::moveOrAttack(int dir)
{
    m_age++;
    int r = m_row;
    int c = m_col;
    if (m_arena->determineNewPosition(r, c, dir))
    {
        if (m_arena->nRobotsAt(r, c) > 0)
            m_arena->attackRobotAt(r, c, dir);
        else
        {
            m_row = r;
            m_col = c;
        }
    }
}

bool Player::isDead() const
{
    return m_dead;
}

void Player::setDead()
{
```

```
    m_dead = true;
}

// Robot.h

#ifndef ROBOT_INCLUDED
#define ROBOT_INCLUDED

class Arena;

class Robot
{
  public:
        // Constructor
    Robot(Arena* ap, int r, int c);

        // Accessors
    int  row() const;
    int  col() const;

        // Mutators
    void move();
    bool getAttacked(int dir);

  private:
    Arena* m_arena;
    int    m_row;
    int    m_col;
    int    m_health;
};

#endif // ROBOT_INCLUDED

// Robot.cpp

#include "Robot.h"
#include "Arena.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Robot::Robot(Arena* ap, int r, int c)
 : m_arena(ap), m_row(r), m_col(c), m_health(INITIAL_ROBOT_HEALTH)
{
    if (ap == nullptr)
    {
        cout << "***** A robot must be in some Arena!" << endl;
```

```
        exit(1);
    }
    if (r < 1  ||  r > ap->rows()  ||  c < 1  ||  c > ap->cols())
    {
        cout << "***** Robot created with invalid coordinates (" << r << ","
             << c << ")!" << endl;
        exit(1);
    }
}

int Robot::row() const
{
    return m_row;
}

int Robot::col() const
{
    return m_col;
}

void Robot::move()
{
      // Attempt to move in a random direction; if we can't move, don't move
    int dir = randInt(0, 3);  // dir is now UP, DOWN, LEFT, or RIGHT
    m_arena->determineNewPosition(m_row, m_col, dir);
}

bool Robot::getAttacked(int dir)  // return true if dies
{
    m_health--;
    if (m_health == 0)
        return true;
    if ( ! m_arena->determineNewPosition(m_row, m_col, dir))
    {
        m_health = 0;
        return true;
    }
    return false;
}
```

==========

HOMEWORK 2

==========

```
  #include <stack>
```

```
        using namespace std;

        int main()
        {
            stack<Coord> coordStack;     // declare a stack of Coords

            Coord a(5,6);
            coordStack.push(a);          // push the coordinate (5,6)
            coordStack.push(Coord(3,4)); // push the coordinate (3,4)
            ...
            Coord b = coordStack.top();  // look at top item in the stack
            coordStack.pop();            // remove the top item from stack
            if (coordStack.empty())      // Is the stack empty?
                cout << "empty!" << endl;
            cout << coordStack.size() << endl;  // num of elements
        }

 #include <queue>
        using namespace std;

        int main()
        {
            queue<Coord> coordQueue;    // declare a queue of Coords

            Coord a(5,6);
            coordQueue.push(a);            // enqueue item at back of queue
            coordQueue.push(Coord(3,4));   // enqueue item at back of queue
            ...
            Coord b = coordQueue.front();  // look at front item
            coordQueue.pop();              // remove the front item from queue
            if (coordQueue.empty())        // Is the queue empty?
                cout << "empty!" << endl;
            cout << coordQueue.size() << endl;  // num of elements
        }
Spring 2016 CS 32
Homework 2 Solution

Problem 1       Problem 2       Problem 3       Problem 4       Problem 5
Problem 1:

#include <stack>

using namespace std;

const char WALL = 'X';
const char OPEN = '.';
const char SEEN = 'o';
```

```
class Coord
{
  public:
    Coord(int rr, int cc) : m_r(rr), m_c(cc) {}
    int r() const { return m_r; }
    int c() const { return m_c; }
  private:
    int m_r;
    int m_c;
};

void explore(char maze[][10], stack<Coord>& toDo, int r, int c)
{
        if (maze[r][c] == OPEN)
        {
             toDo.push(Coord(r,c));
             maze[r][c] = SEEN;  // anything non-OPEN will do
        }
}

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    if (sr < 0  ||  sr > 9  ||  sc < 0  || sc > 9  ||
        er < 0  ||  er > 9  ||  ec < 0  || ec > 9  ||
        maze[sr][sc] != OPEN  ||  maze[er][ec] != OPEN)
       return false;

    stack<Coord> toDo;
    explore(maze, toDo, sr, sc);

    while ( ! toDo.empty() )
    {
        Coord curr = toDo.top();
        toDo.pop();

        const int cr = curr.r();
        const int cc = curr.c();

        if (cr == er  &&  cc == ec)
            return true;

        explore(maze, toDo, cr-1, cc);  // north
        explore(maze, toDo, cr, cc+1);  // east
        explore(maze, toDo, cr+1, cc);  // south
        explore(maze, toDo, cr, cc-1);  // west
    }
    return false;
}
```

Problem 2:

(6,4) (6,5) (7,5) (8,5) (8,6) (8,7) (8,8) (7,8) (6,8) (6,6) (5,4)

Problem 3:

Make three changes to the Problem 1 solution:

Change #include <stack> to #include <queue>

Change stack<Coord> to queue<Coord>

Change Coord curr = toDo.top(); to Coord curr = toDo.front();

Problem 4:

(6,4) (5,4) (6,5) (4,4) (6,6) (7,5) (3,4) (4,5) (8,5) (2,4) (4,6)

The stack solution visits the cells in a depth-first order: it continues along a path until it hits a dead end, then backtracks to the most recently visited intersection that has unexplored branches. Because we're using a stack, the next cell to be visited will be a neighbor of the most recently visited cell with unexplored neighbors.

The queue solution visits the cells in a breadth-first order: it visits all the cells at distance 1 from the start cell, then all those at distance 2, then all those at distance 3, etc. Beacuse we're using a queue, the next cell to be visited will be a neighbor of the least recently visited cell with unexplored neighbors.

Problem 5:

```
// eval.cpp

#include <string>
#include <stack>
#include <cctype>
#include <cassert>
using namespace std;

inline
bool isLetterOrCloseParen(char ch)
{
    return ch == 'T' || ch == 'F' || ch == ')';
}

inline
int precedence(char ch)
  //  Precondition:  ch is in "|&!("
{
    static string ops = "|&!(";
```

```
    static int prec[4] = { 1, 2, 3, 0 };
    int pos = ops.find(ch);
    assert(pos != string::npos);  // must be found!
    return prec[pos];
}

const int RET_OK_EVALUATION      = 0;
const int RET_INVALID_EXPRESSION = 1;

int evaluate(string infix, string& postfix, bool& result)
  // Evaluates a boolean expression
  //   If infix is a syntactically valid infix boolean expression, then set
  //   postfix to the postfix form of that expression, set result to the value
  //   of the expression, and return zero.  If infix is not a syntactically
  //   valid expression, return 1; in that case, postfix may or may not be
  //   changed and but result must be unchanged.
{
      // First convert infix to postfix

    postfix = "";
    stack<char> operatorStack;
    char prevch = '|';  // pretend the previous character was an operator

    for (size_t k = 0; k < infix.size(); k++)
    {
        char ch = infix[k];
        switch(ch)
        {
          case ' ':
            continue;  // do not set prevch to this char

          case 'T':
          case 'F':
            if (isLetterOrCloseParen(prevch))
                return RET_INVALID_EXPRESSION;
            postfix += ch;
            break;

          case '(':
          case '!':
            if (isLetterOrCloseParen(prevch))
                return RET_INVALID_EXPRESSION;
            operatorStack.push(ch);
            break;

          case ')':
            if ( ! isLetterOrCloseParen(prevch))
                return RET_INVALID_EXPRESSION;
```

```
        for (;;)
        {
            if (operatorStack.empty())
                return RET_INVALID_EXPRESSION;  // too many ')'
            char c = operatorStack.top();
            operatorStack.pop();
            if (c == '(')
                break;
            postfix += c;
        }
        break;

      case '|':
      case '&':
        if ( ! isLetterOrCloseParen(prevch))
            return RET_INVALID_EXPRESSION;
        while ( ! operatorStack.empty()  &&
                    precedence(ch) <= precedence(operatorStack.top()) )
        {
            postfix += operatorStack.top();
            operatorStack.pop();
        }
        operatorStack.push(ch);
        break;

      default:  // bad char
        return RET_INVALID_EXPRESSION;
    }
    prevch = ch;
}

  // end of expression; pop remaining operators

if ( ! isLetterOrCloseParen(prevch))
    return RET_INVALID_EXPRESSION;
while ( ! operatorStack.empty())
{
    char c = operatorStack.top();
    operatorStack.pop();
    if (c == '(')
        return RET_INVALID_EXPRESSION;  // too many '('
    postfix += c;
}
if (postfix.empty())
    return RET_INVALID_EXPRESSION;  // empty expression

  // postfix now contains the converted expression
  // Now evaluate the postfix expression
```

```
    stack<bool> operandStack;
    for (size_t k = 0; k < postfix.size(); k++)
    {
        char ch = postfix[k];
        if (ch == 'T')
            operandStack.push(true);
        else if (ch == 'F')
            operandStack.push(false);
        else
        {
            bool opd2 = operandStack.top();
            operandStack.pop();
            if (ch == '!')
                operandStack.push(!opd2);
            else
            {   bool opd1 = operandStack.top();
                operandStack.pop();
                if (ch == '&')
                    operandStack.push(opd1 && opd2);
                else if (ch == '|')
                    operandStack.push(opd1 || opd2);
                else  // Impossible!
                    return RET_INVALID_EXPRESSION;  // pretend it's an invalid
expression
            }
       }
    }
    if (operandStack.size() != 1)  // Impossible!
        return RET_INVALID_EXPRESSION;  // pretend it's an invalid expression
    result = operandStack.top();

    return RET_OK_EVALUATION;
}

// Here's an interactive test driver:
// #include <iostream>
// #include <string>
// using namespace std;
//
// int main()
// {
//     string s;
//     while (getline(cin,s)  &&  s != "exit")
//     {
//         string postfix;
//         bool val;
//         switch (evaluate(s, postfix, val))
```

```
//         {
//           case RET_OK_EVALUATION:
//             cout << "Postfix is " << postfix << " and value is "
//                  << (val ? "true" : "false") << endl;
//             break;
//           case RET_INVALID_EXPRESSION:
//             cout << "Malformed expression" << endl;
//             break;
//           default:
//             cout << "Impossible return code" << endl;
//             break;
//         }
//     }
// }
```

CLASS NOTES

==========

sorting algorithm/ recursion methods

==========

```
sort (an unsorted pile of N items)
{
        if (N > 1)
        {
        split the pile into two unsorted subpiles ofabout N/2 items each
        sort (left subpile)
        sort(right subpile)
        merge the two subpiles into one sorted pile
        }
}
```

base cases : (N <=1)
paths through the functions that does not make any recursive calls

recursive cases:
show that if you assume the function works for problems coser to the base case, then the
function works

ex.

sort 4
adcb

sort 2
ad

```
sort1   sort1
a           d

merge2
ad

sort 2
cb

sort1   sort1
c        b

merge2

bc

merge4
abcd

void sort (int a[], int b, int e)
{
        if (e-b > 1)
        {
        int mid = (b+e)/2;
        sort(a,b,mid);
        sort(a,mid,e);
        merge the two sorted subpiles into one sorted pile
        }
}


int main()
{
        int arr[5] = {50,20,30,10,40};
        sort (arr, 0,5);
}

//

bool contains(int a[], int n, int target)
{
        if (n <=0)
        return false;
        if (a[0] == target)
        return true;
        contains(a+1,n-1,target);
}
```

abstract class
==========
definition for Figure::draw , even though it could be trivial.
If you make the member function Figure::draw a pure virtual function , then
you do not need to give any definition to that member function. The way to make a
member function into a pure virtual function is to mark it as virtual and to add the
anno tation = 0 to the member function declaration, as in the following

example: virtual void draw( ) = 0;

virtual void printCheck( ) const = 0;

Any kind of member can be made a pure virtual function. It need not be a void
function with no parameters as in our example.
A class with one or more pure virtual functions is called an abstract class . An
abstract class can only be used as a base class to derive other classes. You cannot
create
objects of an abstract class, since it is not a complete class definition. An
abstract class
is a partial class definition because it can contain other member functions that are
not pure virtual functions. An abstract class is also a type, so you can write code
with
parameters of the abstract class type and it will apply to all objects of classes
that are
descendants of the abstract class.
If you derive a class from an abstract class, the derived class will itself be an
abstract
class unless you provide definitions for all the inherited pure virtual functions
(and also
do not introduce any new pure virtual functions). If you do provide definitions for
all
the inherited pure virtual functions (and also do not introduce any new pure virtual
functions), the resulting class is not an abstract class, which means you can create
objects of the class.

member iniailization list
==========
initialization section

```
DayOfYear::DayOfYear(int monthValue, int dayValue): month(monthValue), day(dayValue)
{
...
}
```

for inheritance

employee is the base class

```
HourlyEmployee::HourlyEmployee(const string&theName,
const string& theNumber, double theWageRate,
double theHours)
: Employee(theName, theNumber),
wageRate(theWageRate), hours(theHours)
{
//deliberately empty
}
```

default constructor

```
HourlyEmployee::HourlyEmployee( ) : Employee( ), wageRate(0),
hours(0)
{
//deliberately empty
}
```

virtual
==========
Since bill was declared virtual in the base class,
it is automatically virtual in the derived class
DiscountSale . You can add the modifier virtual
to the declaration of bill or omit it as here; in either
case bill is virtual in the class DiscountSale .
(We prefer to include the word virtual in all virtual
function declarations, even if it is not required. We
omitted it here to illustrate that it is not required.)

NOTE:You do not repeat the
qualifier virtual in
the function definition.

virtual destructors
======================

If your derived class destructor is virtual then objects will be destrcuted in a order(firstly derived object then base). If your derived class destructor is NOT virtual then only base class object will get deleted(because pointer is of base class "Base *myObj"). So there will be memory leak for derived object.

To sum up, always make base classes' destructors virtual when they're meant to be manipulated polymorphically.

You also NEED to implement the destructor:

```
class A {
public:
    virtual ~A() = 0;
};

inline A::~A() { }
```

should suffice.

If you derive anything from A and then try to delete or destroy it, A's destructor will eventually be called. Since it is pure and doesn't have an implementation, undefined behavior will ensue. On one popular platform, that will invoke the purecall handler and crash.

ex

```
class base
{

public:
    base(){cout<<"Base Constructor Called\n";}
    virtual ~base(){cout<<"Base Destructor called\n";}

};
class derived1:public base
{

public:
    derived1(){cout<<"Derived constructor called\n";}
    ~derived1(){cout<<"Derived destructor called\n";}

};
int main()
{

    base* b;
    b=new derived1;
    delete b;

}
```

different ways of overloading a +

==========

Suppose you have a class like this:

```
class Element {
public:
    Element(int value) : value(value) {}
```

```
    int getValue() const { return value; }
private:
    int value;
};
```

There are four ways to define a binary operator such as +.

(1) As a free function with access to only the public members of the class:

```
// Left operand is 'a'; right is 'b'.
Element operator+(const Element& a, const Element& b)
 {
    return Element(a.getValue() + b.getValue());
}
```

e1 + e2 == operator+(e1, e2)

(2) As a member function, with access to all members of the class:

```
class Element
{
public:
    // Left operand is 'this'; right is 'other'.
    Element operator+(const Element& other) const {
        return Element(value + other.value);
    }
    // ...
};
```

e1 + e2 == e1.operator+(e2)

(3) As a friend function, with access to all members of the class:

```
class Element
{
public:
    // Left operand is 'a'; right is 'b'.
    friend Element operator+(const Element& a, const Element& b) {
        return a.value + b.value;
    }
    // ...
};
```

e1 + e2 == operator+(e1, e2)

(4) As a friend function defined outside the class body—identical in behaviour to #3:

```
class Element
{
public:
    friend Element operator+(const Element&, const Element&);
    // ...
};
```

```
Element operator+(const Element& a, const Element& b)
{
    return a.value + b.value;
}
e1 + e2 == operator+(e1, e2)
=================================================
bool contains(const int a[], int n, int target)
{
if (n <= 0)
return false;
if (a[0] == target
return true;
return contains(a+1,n-1,target);
}
```

Templates
==========

overloading functions with different parameters

what if you want the same functions but for different types?

ex.

```
template(typename T>

T mininum(T a, T b)
{
if (a < b)
return a;
else
return b;
}
```

How do we use this?

ex.

```
cout << mininum<int>(k,10)<<endl;
//almost never done explicitly
//so.....
```

"template argument deduction"
have complier deduce what type it is
so the compiler changes T for you when function is called

instantiated template must compile,
so if the two arguments is the same type, and you pass in a int and a double, it

won't work

instantiated template must do the right thing
ex.

```
template<typename T, typename U>

T mininum(T a, U b)
{
....
}
```

what if T was int and U was double, and we pass in 3.5, then it will return 3 instead, which is wrong
even though it compiles

T() calls the default constructor or a built in type like string() or double()

```
T sum(const T a[], int n)
{
        T total = T();
for (int k =0; k < n; k++)
        total += a[k];
        return total;
}
```

what if you pass two arrays of char into the mininum function,
will convert T to pointer of char
will not work right because you will be comparing pointers not what it is pointing to

```
//must overload function

char* mininum(char* a, char* b)
{
if (strcmp(a,b) < 0)
return a;
else
return b;
}
//non template function will be called instead of the template one
```

template class
==========

```
template<typename T>
class Stack
```

```
{
public:
stack();
void push(Const T& x);
void pop();
T top() const;
int size() const;

private:

T m_data[100];
int m_top;
};

template<typename T>
stack<T>::stack() : m_top(0)
{}

template<typename T>
void stack<T>::push(const  T& x)
{
m_data[m_top] = x;
m_top++;
}

template<typename T>
void stack<T>::pop()
{
m_top--;
}

template<typename T>
T stack<T>:: top() const
{
return m_data[m_top-1];
}

template<typename T>
int stack<T>::size() const
{
return m_top;
}


// you must put template<typename T> before function implementation or class
declaration
```

iterators
==========

```
template<typename Iter>
Iter findFirstNegative(Iter b, Iter e)
{
for ( ; b != e; b++)
if (*b < 0)
break;
return b;
}
template<typename Iter>
Iter findFirstEmpty(Iter b, Iter e)
{
for ( ; b != e; b++)
if (b->empty())
break;
return b;
}
```

pointers to functions
==========

```
// fcnPtr is a pointer to a function that takes no arguments and returns an integer
int (*fcnPtr)();
```

Assigning a function to a function pointer

```
int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    int (*fcnPtr)() = foo; // fcnPtr points to function foo
    fcnPtr = goo; // fcnPtr now points to function goo

    return 0;
}
```

```
// function prototypes
int foo();
double goo();
int hoo(int x);

// function pointer assignments
int (*fcnPtr1)() = foo; // okay
int (*fcnPtr2)() = goo; // wrong -- return types don't match!
double (*fcnPtr4)() = goo; // okay
fcnPtr1 = hoo; // wrong -- fcnPtr1 has no parameters, but hoo() does
int (*fcnPtr3)(int) = hoo; // okay
```

Calling a function using a function pointer

The other primary thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    (*fcnPtr)(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

The second way is via implicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    fcnPtr(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

As you can see, the implicit dereference method looks just like a normal function call -- which is what you'd expect, since normal function names are pointers to

functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

One interesting note: Default parameters won't work with function pointers. Default arguments are resolved at compile-time (that is, if you don't supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time. Consequently, default parameters can not be resolved when making a function call with a function pointer. You'll explicitly have to pass in values for any defaulted parameters in this case.

Passing functions as arguments to other functions

```
bool (*comparisonFcn)(int, int);

void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
// Step through each element of the array
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so
far.
        int smallestIndex = startIndex;

        // Look for smallest element remaining in the array (starting at
startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (comparisonFcn(array[smallestIndex], array[currentIndex])) //
COMPARISON DONE HERE
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}


bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

int main()
```

```
{
int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

selectionSort(array, 9, ascending);
}

note: log base 2
assume T(N) = N*log(N)

T(N) = 2*T(N/2) + N
T(N) = 2*(N/2*log(N/2))+N
=N*(log(N)-log(2)) + N
=N*log(N) - N + N
=N*log(N)

merge sort is
O(NlogN)

selection sort
O(N^2)

bubble sort
worst: O(N^2)
average O(N^2)
best O(N)

insertion sort
worstcase
worst: O(N^2)
average O(N^2)
best O(N)


quick sort

Sort(N items)
{
if(N>2)
{
partition into 2 piles-"early" and "late"
sort early
sort late
stack the early pile onto the late pile
}
}

quick sort
        average O(NlogN)
```

better constant of proportionality than merge sort

trees
==========
root node
parent/child
subtree
path
leaf nodes
interior nodes
-all nodes with child is a interior node
depth of a node
-root node is at depth zero
height of a tree
-so the first child is at depth one

```
struct Node
{
string data;
vector<Node*> children
}

Node* root;


int count (const Node* t)
{
if ( t == nullptr)
return 0;
return countAuxilliary(t);
}
int countAuxilliary(const Node* t) // t must not be null
{

int total = 1;
for (int k = 0; k!= t->children.size(); k++)
        total += countAuxillary(t-> children[k]);
retur total;
}

void printTreeAuxilliary(const Node* t, int depth)
{
if ( t != nullptr)
{
cout << string (2*depth,' ') << t->data << endl;
for(int k = 0; k!= t->children[k].size(); k++)
    printTree(t->children[k], depth+1);
}
```

```
}

void printTree(const Node* t)
{
printTreeAuxilliary(t,0);
}


//if you want to not have an auxiliary function use default arguments
void printTree(const Node* t, int depth = 0);

printTree(root);
```

==========

HOMEWORK 4

==========
=

```
// Set.h

#ifndef SET_INCLUDED
#define SET_INCLUDED

template<typename ItemType>
class Set
{
public:
        Set();               // Create an empty set.
        bool empty() const;  // Return true if the set is empty, otherwise false.
        int size() const;    // Return the number of items in the set.

        bool insert(const ItemType& value);
        // Insert value into the set if it is not already present.  Return
        // true if the value was actually inserted.  Leave the set unchanged
        // and return false if the value was not inserted (perhaps because it
        // is already in the set or because the set has a fixed capacity and
        // is full).

        bool erase(const ItemType& value);
        // Remove the value from the set if present.  Return true if the
        // value was removed; otherwise, leave the set unchanged and
        // return false.

        bool contains(const ItemType& value) const;
        // Return true if the value is in the set, otherwise false.
```

```
        bool get(int i, ItemType& value) const;
        // If 0 <= i < size(), copy into value an item in the set and
        // return true.  Otherwise, leave value unchanged and return false.

        void swap(Set& other);
        // Exchange the contents of this set with the other one.

        // Housekeeping functions
        ~Set();
        Set(const Set& other);
        Set& operator=(const Set& rhs);

private:
        // Representation:
        //   a circular doubly-linked list with a dummy node.
        //   m_head points to the dummy node.
        //   m_head->m_prev->m_next == m_head and m_head->m_next->m_prev == m_head
        //   m_size == 0  iff  m_head->m_next == m_head->m_prev == m_head

        struct Node
        {
                ItemType m_value;
                Node*    m_next;
                Node*    m_prev;
        };

        Node* m_head;
        int   m_size;

        void createEmpty();
        // Create an empty list.  (Will be called only by constructors.)

        void insertAtTail(const ItemType& value);
        // Insert value in a new Node at the tail of the list, incrementing
        // m_size.

        void doErase(Node* p);
        // Remove the Node p, decrementing m_size.

        Node* find(const ItemType& value) const;
        // Return pointer to Node whose m_value == value if present, else m_head
};

// Declarations of non-member functions

template<typename ItemType>
void unite(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>& result);
// result = { x | (x in s1) OR (x in s2) }
```

```
template<typename ItemType>
void subtract(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result);
// result = { x | (x in s1) AND NOT (x in s2) }

// Inline implementations

template<typename ItemType>
int Set<ItemType>::size() const
{
        return m_size;
}

template<typename ItemType>
bool Set<ItemType>::empty() const
{
        return size() == 0;
}

template<typename ItemType>
bool Set<ItemType>::contains(const ItemType& value) const
{
        return find(value) != m_head;
}

// Set.cpp

template<typename ItemType>
Set<ItemType>::Set()
{
        createEmpty();
}

template<typename ItemType>
bool Set<ItemType>::insert(const ItemType& value)
{
        // Fail if value already present

        if (contains(value))
                return false;

        // Insert new Node (at tail; choice of position is arbitrary),
        // incrementing m_size

        insertAtTail(value);
        return true;
}
```

```
template<typename ItemType>
bool Set<ItemType>::erase(const ItemType& value)
{
        // Find the Node with the value, failing if there is none.

        Node* p = find(value);
        if (p == m_head)
                return false;

        // Erase the Node, decrementing m_size
        doErase(p);
        return true;
}

template<typename ItemType>
bool Set<ItemType>::get(int i, ItemType& value) const
{
        if (i < 0 || i >= m_size)
                return false;

        // Return the value at position i.  This is one way of ensuring the
        // required behavior of get:  If the Set doesn't change in the interim,
        // * calling get with each i in 0 <= i < size() gets each of the
        //   Set elements, and
        // * calling get with the same value of i each time gets the same element.

        // If i is closer to the head of the list, go forward to reach that
        // position; otherwise, start from tail and go backward.

        Node* p;
        if (i < m_size / 2)  // closer to head
        {
                p = m_head->m_next;
                for (int k = 0; k != i; k++)
                        p = p->m_next;
        }
        else  // closer to tail
        {
                p = m_head->m_prev;
                for (int k = m_size - 1; k != i; k--)
                        p = p->m_prev;
        }

        value = p->m_value;
        return true;
}
```

```
template<typename ItemType>
void Set<ItemType>::swap(Set<ItemType>& other)
{
        // Swap head pointers

        Node* p = other.m_head;
        other.m_head = m_head;
        m_head = p;

        // Swap sizes

        int s = other.m_size;
        other.m_size = m_size;
        m_size = s;
}

template<typename ItemType>
Set<ItemType>::~Set()
{
        // Delete all Nodes from first non-dummy up to but not including
        // the dummy

        while (m_head->m_next != m_head)
                doErase(m_head->m_next);

        // delete the dummy

        delete m_head;
}

template<typename ItemType>
Set<ItemType>::Set(const Set& other)
{
        createEmpty();

        // Copy all non-dummy other Nodes.  (This will set m_size.)
        // Inserting each new node at the tail rather than anywhere else is
        // an arbitrary choice.

        for (Node* p = other.m_head->m_next; p != other.m_head; p = p->m_next)
                insertAtTail(p->m_value);
}

template<typename ItemType>
Set<ItemType>& Set<ItemType>::operator=(const Set& rhs)
{
        if (this != &rhs)
        {
```

```
                // Copy and swap idiom

                Set temp(rhs);
                swap(temp);
        }
        return *this;
}

template<typename ItemType>
void Set<ItemType>::createEmpty()
{
        m_size = 0;

        // Create dummy node

        m_head = new Node;
        m_head->m_next = m_head;
        m_head->m_prev = m_head;
}

template<typename ItemType>
void Set<ItemType>::insertAtTail(const ItemType& value)
{
        // Create a new node

        Node* newNode = new Node;
        newNode->m_value = value;

        // Insert new item at tail of list (predecessor of the dummy at m_head)
        //     Adjust forward links

        newNode->m_next = m_head;
        m_head->m_prev->m_next = newNode;

        //     Adjust backward links

        newNode->m_prev = m_head->m_prev;
        m_head->m_prev = newNode;

        m_size++;
}

template<typename ItemType>
void Set<ItemType>::doErase(Node* p)
{
        // Unlink p from the list and destroy it

        p->m_prev->m_next = p->m_next;
```

```
        p->m_next->m_prev = p->m_prev;
        delete p;

        m_size--;
}

template<typename ItemType>
typename Set<ItemType>::Node* Set<ItemType>::find(const ItemType& value) const
{
        // Walk through the list looking for a match

        Node* p = m_head->m_next;
        for (; p != m_head && p->m_value != value; p = p->m_next)
                ;
        return p;
}


template<typename ItemType>
void unite(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>& result)
{
        // Check for aliasing to get correct behavior or better performance:
        // If result is s1 and s2, result already is the union.
        // If result is s1, insert s2's elements into result.
        // If result is s2, insert s1's elements into result.
        // If result is a distinct set, assign it s1's contents, then
        //   insert s2's elements in result, unless s2 is s1, in which
        //   case result now already is the union.

        const Set<ItemType>* sp = &s2;
        if (&result == &s1)
        {
                if (&result == &s2)
                        return;
        }
        else if (&result == &s2)
                sp = &s1;
        else
        {
                result = s1;
                if (&s1 == &s2)
                        return;
        }
        for (int k = 0; k < sp->size(); k++)
        {
                ItemType v;
                sp->get(k, v);
                result.insert(v);
```

```
        }
}

template<typename ItemType>
void subtract(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result)
{
        // Guard against the case that result is an alias for s2 by copying
        // s2 to a local variable.  This implementation needs no precaution
        // against result being an alias for s1.

        Set<ItemType> s2copy(s2);
        result = s1;
        for (int k = 0; k < s2copy.size(); k++)
        {
                ItemType v;
                s2copy.get(k, v);
                result.erase(v);
        }
}

#endif // SET_INCLUDED
```