Vector class

// for a vector of Items use Vector<Item>, e.g., Vector<int> intvector;
/
//
// note: Item must have a default constructor
//
// constructors:
// Vector() -- default, vector of size 0 (no entries)
// Vector(int size) -- vector with size entries
// Vector(int size,
// Item fillValue) -- vector w/ size entries all == fillValue
// Vector(const Vector & vec) -- copy constructor
//
// int Length() -- returns size of vector (capacity)
// void SetSize(int newSize) -- resizes the vector to newSize elements
// (can result in losing elements if
// new size < old size)
// void resize(int newSize) -- synonym for SetSize
//
// void Fill(Item fillValue) -- set all entries == fillValue
//
// operator = -- assignment operator works properly
// operator [] -- indexes both const and non-const vectors
//
/
// examples of use:
//
// Vector<double> dlist(100); // a list of 100 doubles
// Vector<double> dzlist(100,0.0); // initialized to 0.0
//
// Vector<String> slist(300); // 300 strings
//
// Vector<int> ilist; // has room for 0 ints

Matrix class

// for a matrix of Items use Matrix<Item>, e.g., Matrix<int> intmatrix;
/
//
// note: Item must have a default constructor
//
// constructors:
// Matrix() -- default, matrix of size 0x0
// Matrix(int rows,int cols) -- matrix with dimensions rows x cols
// Matrix(int rows,int cols,
// Item fillValue) -- matrix w/ entries all == fillValue
Matrix(const Matrix & vec) -- copy constructor

int Rows() -- returns # of rows (capacity)
int Cols() -- returns # of columns (capacity)

void SetSize(int rows, int cols) -- resizes the matrix to rows x cols
    can lose entries if size is smaller
    in either dimension

void Fill(Item fillValue) -- set all entries == fillValue

operator = -- assignment operator works properly
operator [] -- indexes const and non-const matrixs

examples of use:
Matrix<double> dmat(100,80);    // 100 x 80 matrix of doubles
Matrix<double> dzmat(100,80,0.0); // initialized to 0.0
Matrix<String> smat(300,1);    // 300 strings
Matrix<int> imat;              // has room for 0 ints

PROBLEM 1:  (Baseball is alive again: (25 pts))

Consider the following definition, BaseBall iterator class, and constructor. The BaseBall iterator
class contains information on all the baseball teams in a league. In particular, it iterates over all
the players in the league and for each player gives information about the player (name and number
of homeruns hit) and the name of the team the player is on.

struct Player
{
    string name;  // name of baseball player
    int numHRs;   // number of homeruns hit this season

    void Print(); // print name of baseball player and number of HRs hit
};

class BaseBall
{
public:
    BaseBall();  // constructor
    BaseBall(int teams, int playercount); // constructor

    // iterator
    void First();  // iterator set to first player
void Next(); // iterator set to next player
Player Current(); // returns name and number of HRs for current player
string CurrentTeam(); // returns name of team for the current player
bool IsDone(); // returns true if iterator doesn’t have
// a current player, returns false otherwise

void Print() const; // prints teams and players

private:

void Read(); // reads in team names and player info

Vector <string> myTeamNames; // names of teams
Matrix <Player> myTeamPlayers; // names and number of HRs for players,
// each row represents one team
int myNumTeams; // number of teams
int myNumPlayersPerTeam; // number of players per team
int myCurrentRow; // current row in iterator
int myCurrentCol; // current column in iterator

BaseBall::BaseBall(int teams, int playercount) // constructor
: myTeamNames(teams), myTeamPlayers(teams, playercount),
  myNumTeams(teams), myNumPlayersPerTeam(playercount),
  myCurrentRow(0), myCurrentCol(0)
{
  Read(); // code not shown, reads in team names and player info
}

Here is a sample code segment that uses the BaseBall class to list out all the teams and for each
team to list the players on that team and the number of homeruns they have hit this season. Then
it prints a blank line followed by the name and number of homeruns of the first player. Note for
the purposes of testing, we are using a small data file with just 4 teams and 5 players.

BaseBall year98(4,5);
year98.Print(); // list out all teams and players
cout << endl;
year98.First();
year98.Current().Print(); // list out first player and number of homeruns

Here is the corresponding output. On each row is the team name followed by players and the
number of homeruns for each player, then the first player is printed again.

Mets Lotze 21 Sharma 4 Hanna 8 Allen 33 Fath 12
Reds Krauss 43 Ptak 22 Oka 17 Wong 2 Bostrom 70
Cards Caton 18 McGuire 70 Byrd 3 Fan 28 Sun 11
Cubs Zeng 4 Liu 9 Meng 16 Sosa 66 Finley 3
The function *Read* reads in information about the baseball teams. The name of a baseball team is stored in the vector *myTeamNames*, and information about the corresponding players on that team are stored in the matrix *myTeamPlayers* in the same row number. The team names are stored in no particular order, and the players on a team are stored in no particular order. You may assume *Read* already exists, you DO NOT need to write it.

Complete the following functions and code segment below. Note that the iterator moves over all the players from all the teams and at any point must be able to return a current player (function *Current*) and the name of the team the player is on (function *CurrentTeam*). Do NOT add any additional private variables to the *BaseBall* class.

- Complete the function *First* below (3 pts).

```cpp
void BaseBall::First()
// postcondition: iterator set to first player
{
}
```

- Complete the function *IsDone* below (3 pts).

```cpp
bool BaseBall::IsDone()
// postcondition: returns true if iterator has no current player,
// otherwise returns false
{
}
```

- Complete the function *Next* below (5 pts).

```cpp
void BaseBall::Next()
// postcondition: iterator moved to next player
{
}
```

- Complete the function *Current* below (3 pts).

```cpp
Player BaseBall::Current()
// postcondition: returns current player in iterator
{
}
```
Complete the function `CurrentTeam` below (3 pts).

```cpp
string Baseball::CurrentTeam()
// postcondition: returns current team in iterator
{
}
```

For this part, complete code that is part of a client program using the `BaseBall` class. Compute the maximum number of homeruns hit by a player and the list of names hitting that many homeruns.

For example, if the code segment was applied to the data on page 5, the output would be:

70 homeruns
McGuire Bostrom

Complete the code segment below (6 pts).

```cpp
Baseball database(M,N); // read in info for M teams, there are
             // N players per team
```

Assume that there are M teams and N players per team. What is the worst case running time (big-Oh) of the client program code segment you wrote above (do not count the time needed for the constructor, which reads in the info, just analyze the code you write) (2 pts)?

**PROBLEM 2:** (Animal Business: (25 pts))

Consider the following definition to keep track of the type and number of each type of animals in a zoo.

```cpp
struct Critter
{
    string type;        // type of animal
    int number;         // number of this type of animal

    Critter(string, int);  // constructor
};

Critter::Critter(string ty, int num)
: type(ty), number(num)
{
}
```

page 5
PART A (5 pts):
Write the function \textit{Read} whose header is shown below. \textit{Read} reads data from an input stream consisting of pairs of a word and number (type of animal and number of this type) until there is no more data. Information for each animal is read in and stored in the vector in the order it is read in.

For example, if the stream \textit{input} contains the information:

\begin{itemize}
  \item elephant 3
  \item zebra 5
  \item hippo 2
  \item elephant 3
  \item elephant 2
  \item coyote 3
  \item hippo 10
\end{itemize}

Then after the call \textit{Read(NYzoo, numTypes, input)}, \textit{numTypes} will be set to 7, and \textit{NYzoo} will look like:

\begin{center}
\begin{tabular}{cccccccc}
  NYzoo & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
  \hspace*{1em} elephant & 3 & & & & & & \\
  \hspace*{1em} hippo & & 2 & & & & & \\
  \hspace*{1em} elephant & & & 2 & & & & \\
  \hspace*{1em} coyote & & & 3 & & & & \\
  \hspace*{1em} zebra & 5 & & & & & & \\
  \hspace*{1em} elephant & & & & & & & \\
  \hspace*{1em} hippo & & & & & & & 10 \\
\end{tabular}
\end{center}

Complete function \textit{Read} below the following header. You can assume there is enough capacity in \textit{zoo} and that you do not need to resize it.

\begin{verbatim}
void Read( Vector <Critter *> & zoo, int & size, istream & input)
    // precondition: zoo is empty, but has plenty of capacity for the
    // data in the stream input, input is open and bound to a file
    // postcondition: zoo contains data from the stream input and
    // size indicates the number of items stored in zoo
{
    
}
\end{verbatim}

PART B (2 pts): If N is the number of pairs of words and numbers read in, what is the worst case running time (big-Oh) of the \textit{Read} function?

PART C (7 pts):
Write the function \textit{FindDuplicate} whose header is shown below. \textit{FindDuplicate} is given a vector of pointers to \textit{Critters} and returns true if there is a duplicate type in the vector, otherwise it returns false. In addition, if there is a duplicate, \textit{FindDuplicate} sets \textit{pos1} and \textit{pos2} to the index positions of the duplicate. If there are multiple duplicates, \textit{pos1} and \textit{pos2} can be set to any pair of duplicates.
Let \( NYzoo \) represent the vector shown in Part A. The call \( \text{FindDuplicate}(NYzoo, \text{numTypes}, \text{first}, \text{dup}) \) returns true and sets \( \text{first} \) equal to 0 and \( \text{dup} \) equal to 3 (or \( \text{first} \) and \( \text{dup} \) could be set to any of the duplicate pairs).

Complete function \( \text{FindDuplicate} \) below the following header.

```cpp
bool FindDuplicate(Vector <Critter *> zoo, int numTypes, int & pos1, int & pos2)
{
    // precondition: zoo has numTypes items
    // postcondition: returns true if there is a duplicate and sets pos1 and
    //                pos2 to the index positions of the duplicate,
    //                otherwise returns false
    return false;  // Example: No duplicate found
}
```

**PART D (2 pts):** If \( N \) is the number of pairs of words and numbers read in, what is the worst case running time (big-Oh) of the \( \text{FindDuplicate} \) function?

**PART E (9 pts):**
Write the function \( \text{RemoveDuplicates} \) whose header is shown below. \( \text{RemoveDuplicates} \) is given a vector of pointers to \( \text{Critters} \) and combines duplicate types by combining their counts and removing the duplicate entry.

Let \( NYzoo \) represent the vector shown in Part A. After the call \( \text{RemoveDuplicates}(NYzoo, \text{numTypes}) \), numTypes is set to 4 and \( NYzoo \) looks like:

```
NYzoo

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>elephant</td>
<td></td>
<td>hippo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zebra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

In writing \( \text{RemoveDuplicates} \), you may use the function \( \text{FindDuplicate} \) from part C. Assume the function \( \text{FindDuplicate} \) works correctly regardless of what you wrote for part C.

Complete function \( \text{RemoveDuplicates} \) below the following header.

```cpp
void RemoveDuplicates(Vector <Critter *> & zoo, int & numTypes)
{
    // postcondition: all duplicates have combined their counts and the duplicates
    // removed from zoo and numTypes has been adjusted to the current
    // number of items in zoo

    // Example: RemoveDuplicates implementation
}
```

**PROBLEM 3:** (Number Play: (20 pts))
Consider the following definition for a node in a linked list.

```c
struct Node {
    int number;
    Node * next;

    Node(int, Node *);
};
Node::Node(int num, Node * nx) :
    number(num), next(nx)
{};
```

**PART A (4 pts):**
Write the function `Print` whose header is shown below. `Print` outputs the numbers in the given linked list on one line separated by blanks.

For example, consider the linked list numbers shown below.

```
10 25 20 17
```

The output for the call `Print(numbers)` would be:

```
10 25 20 17
```

Complete function `Print` below the following header.

```c
void Print(Node * list)
// postcondition: prints the numbers in the list on one line, separated
// by blanks
{
}
```

**PART B (7 pts):**
Write the function `DoubleNum` whose header is shown below. `DoubleNum` is given a linked list and a number. If the number appears in the linked list, then a copy of the number is inserted into the list immediately after the first occurrence of the number.

For example, if `numbers` is the linked list from Part A, then the first list below is the result of calling `DoubleNum(numbers,25)` and the second list is a result of calling `DoubleNum(numbers,25)` again.
Complete function DoubleNum below the following header.

```c
void DoubleNum(Node * list, int num)
// postcondition: if num appears in list, a copy of the num is inserted
// into the list after the first occurrence of num
{
}
```

**PART C (9 pts):**

Write the function Swap whose header is shown below. Swap is given a linked list and if there are at least two nodes in the list, then the first and last nodes are swapped in the list.

Although Node is rather small, there are eventual plans to add a lot more information to this struct, so the swap should be efficient, which means it should reset pointers in the swap, and NOT swap the contents of the two nodes.

For example, if numbers is the linked list from Part A, then the result of calling Swap(numbers) is shown below.

```c
void Swap(Node * & list)
// postcondition: if list previously had at least two nodes, then the
// first and last nodes of the list have been swapped.
{
}
```