Appendix A

Exercise Solutions

Stanley Lippman contact information:
homepage: www.objectwrite.com
email: slippman@objectwrite.com

AWL: www.aw.com/cseng/series/indepth

Exercise 1.4

Try to extend the program: (1) Ask the user to enter both a first and last name, and (2) modify the output to write out both names.

We need two strings for our extended program: one to hold the user's first name and a second to hold the user's last name. By the end of Chapter 1, we know three ways to support this. We can define two individual string objects:

```
string first_name, last_name;
```

We can define an array of two string objects:

```
string usr_name[ 2 ];
```

Or we can define a vector of two string objects:

```
vector<string> usr_name(2);
```

At this point in the text, arrays and vectors have not yet been introduced, so I've chosen to use the two string objects:

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
      string first_name, last_name;
      cout << "Please enter your first name: ";
      cin >> first_name;

      cout << "hi, " << first_name
           << " Please enter your last name: ";

      cin  >> last_name;
      cout << '\n';
           << "Hello, "
           << first_name  << ' '  << last_name
           << " ... and goodbye!\n";
}
```

When compiled and executed, this program generates the following output (my responses are highlighted in bold):

```
Please enter your first name: stan
hi, stan Please enter your last name: lippman

Hello, stan lippman ... and goodbye!
```

Exercise 1.5

Write a program to ask the user his or her name. Read the response. Confirm that the input is at least two characters in length. If the name seems valid, respond to the user. Provide two implementations: one using a C-style character string, and the other using a string class object.

The two primary differences between a string class object and a C-style character string are that (1) the string class object grows dynamically to accommodate its character string, whereas the C-style character string must be given a fixed size that is (hopefully) large enough to contain the assigned string, and (2) the C-style character string does not know its size. To determine the size of the C-style character string, we must iterate across its elements, counting each one up to but not including the terminating null. The `strlen()` standard library routine provides this service for us:

```
int strlen( const char* );
```

To use `strlen()`, we must include the `cstring` header file.

However, before we get to that, let's look at the string class implementation. Particularly for beginners, I recommend that the string class be used in favor of the C-style character string.

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
    string user_name;

    cout << "Please enter your name: ";
    cin  >> user_name;

    switch ( user_name.size() ){
        case 0:
           cout << "Ah, the user with no name. "
                << "Well, ok, hi, user with no name\n";
           break;

        case 1:
           cout << "A 1-character name? Hmm, have you read Kafka?: "
                << "hello, " << user_name << endl;
            break;

          default:
            // any string longer than 1 character
            cout << "Hello, " << user_name
                 << " -- happy to make your acquaintance!\n";
            break;
      }
      return 0;
}
```

The C-style character string implementation differs in two ways. First, we must decide on a fixed size to declare `user_name`; I've arbitrarily chosen 128, which seems more than adequate. Second, we use the the standard library `strlen()` function to discover the size of `user_name`. The `cstring` header file holds the declaration of `strlen()`. If the user enters a string longer than 127 characters, there will be no room for the terminating null character. To prevent that, I use the `setw()` iostream manipulator to guarantee that we do not read in more than 127 characters. To use the `setw()` manipulator, we must include the `iomanip` header file.

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

int main()
{
    // must allocate a fixed size
    const int nm_size = 128;
    char user_name[ nm_size ];
    cout << "Please enter your name: ";
    cin  >> setw( nm_size ) >> user_name;
```

```
    switch ( strlen( user_name ))
           {
              // same case labels for 0, 1
              case 127:
                  // maybe string was truncated by setw()
                  cout << "That is a very big name, indeed -- "
                       << "we may have needed to shorten it!\n"
                       << "In any case,\n";

                  // no break -- we fall through ...
              default:
                  // the 127 case drops through to here -- no break
                  cout << "Hello, " << user_name
                       << " -- happy to make your acquaintance!\n";
                  break;
           }

    return 0;
}
```

Exercise 1.6

Write a program to read in a sequence of integers from standard input. Place the values, in turn, in a built-in array and a vector. Iterate over the containers to sum the values. Display the sum and average of the entered values to standard output.

The built-in array and the vector class differ in primarily the same ways as the C-style character string (which is implemented as an array of char elements) and the string class: (1) The built-in array must be of a fixed size, whereas the vector can grow dynamically as elements are inserted, and (2) the built-in array does not know its size. The fixed-size nature of the built-in array means that we must be concerned with potentially overflowing its boundary. Unlike the C-style string, the built-in array has no sentinel value (the null) to indicate its end. Particularly for beginners, I recommend that the vector class be used in favor of the built-in array. Here is the program using the vector class:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;
    while ( cin >> ival )
            ivec.push_back( ival );
```

```
    // we could have calculated the sum as we entered the
    // values, but the idea is to iterate over the vector ...
    for ( int sum = 0, ix = 0; ix < ivec.size(); ++ix )
          sum += ivec[ ix ];

    int average = sum / ivec.size();
    cout << "Sum of " << ivec.size()
         << " elements: " << sum
         << ". Average: " << average << endl;
}
```

The primary difference in the following built-in array implementation is the need to monitor the number of elements being read to ensure that we don't overflow the array boundary:

```
#include <iostream>
using namespace std;

int main()
{
    const int array_size = 128;
    int ia[ array_size ];
    int ival, icnt = 0;

    while ( cin >> ival &&
            icnt < array_size )
                 ia[ icnt++ ] = ival;

    for ( int sum = 0, ix = 0; ix < icnt; ++ix )
          sum += ia[ ix ];

    int average = sum / icnt;
    cout << "Sum of " << icnt
         << " elements: " << sum
         << ". Average: " << average << endl;
}
```

Exercise 1.7

Using your favorite editor, type two or more lines of text into a file. Write a program to open the file, reading each word into a `vector<string>` object. Iterate over the vector, displaying it to `cout`. That done, sort the words using the `sort()` generic algorithm.

```
#include <algorithm>
sort( container.begin(), container.end() );
```

Then print the sorted words to an output file.

I open both the input and the output file before reading in and sorting the text. I could wait to open the output file, but what would happen if for some reason the output file failed to open? Then all the computations would have been for nothing. (The file paths are hard-coded and reflect Windows conventions. The `algorithm` header file contains the forward declaration of the `sort()` generic algorithm.)

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <string>
#include <vector>

using namespace std;
int main()
{
    ifstream in_file( "C:\\My Documents\\text.txt"  );
    if ( ! in_file )
       { cerr << "oops! unable to open input file\n"; return -1; }

    ofstream out_file("C:\\My Documents\\text.sort" );
    if ( ! out_file )
       { cerr << "oops! unable to open input file\n"; return -2; }

    string word;
    vector< string > text;
    while ( in_file >> word )
            text.push_back( word );

    int ix;
    cout << "unsorted text: \n";

    for ( ix = 0; ix < text.size(); ++ix )
          cout << text[ ix ] << ' ';
    cout << endl;

    sort( text.begin(), text.end() );

    out_file << "sorted text: \n";
    for ( ix = 0; ix < text.size(); ++ix )
         out_file << text[ ix ] << ' ';
    out_file << endl;

    return 0;
}
```

The input text file consists of the following three lines:

```
we were her pride of ten she named us:
Phoenix, the Prodigal, Benjamin,
and perspicacious, pacific Suzanne.
```

When compiled and executed, the program generates the following output (I've inserted line breaks to display it here on the page):

```
Benjamin, Phoenix, Prodigal, Suzanne.
and her named of pacific perspicacious,
pride she ten the us: we were
```

Exercise 1.8

The `switch` statement of Section 1.4 displays a different consolation message based on the number of wrong guesses. Replace this with an array of four string messages that can be indexed based on the number of wrong guesses.

The first step is to define the array of string messages in which to index. One strategy is to encapsulate them in a display function that, passed the number of incorrect user guesses, returns the appropriate consolation message. Here is a first implementation. Unfortunately, it is not correct. Do you see the problems?

```
const char* msg_to_usr( int num_tries )
{
   static const char* usr_msgs[] = {
      "Oops! Nice guess but not quite it.",
      "Hmm. Sorry. Wrong again.",
      "Ah, this is harder than it looks, isn't it?",
      "It must be getting pretty frustrating by now!"
    };
    return usr_msgs[ num_tries ];
}
```

The index is off by one. If you flip back to the Section 1.4 `switch` statement, you'll see that the number of incorrect tries begins with 1 because, after all, we are responding to wrong guesses on the user's part. Our array of responses, however, begins at position 0. So our responses are always one guess more severe than called for.

There are other problems as well. The user can potentially try more than four times and be wrong with each try, although I capped the number of unique messages at 4. If we unconditionally index into the array, a value of 4 or greater will overflow the array boundary. Moreover, we must guard against other potential invalid values such as a negative number.

Here is a second iteration. I've added a new first message in case the user somehow has not yet guessed. I don't expect we'll actually return it, but in this way, the other messages at least are in their "natural" position. I defined a `const` object to hold a count of the number of entries in the array.

```
const char* msg_to_usr( int num_tries )
{
   const int rsp_cnt = 5;
   static const char* usr_msgs[ rsp_cnt ] = {
```

```
        "Go on, make a guess. ",
        "Oops! Nice guess but not quite it.",
        "Hmm. Sorry. Wrong again.",
        "Ah, this is harder than it looks, no?",
        "It must be getting pretty frustrating by now!"
    };
    if ( num_tries < 0 )
         num_tries = 0;
    else
    if ( num_tries >= rsp_cnt )
         num_tries = rsp_cnt-1;
    return usr_msgs[ num_tries ];
}
```

Exercise 2.1

`main()` [in Section 2.1] allows the user to enter only one position value and then terminates. If a user wishes to ask for two or more positions, he must execute the program two or more times. Modify `main()` [in Section 2.1] to allow the user to keep entering positions until he indicates he wishes to stop.

We use a `while` loop to execute the "solicit position, return value" code sequence. After each iteration, we ask the user whether he wishes to continue. The loop terminates when he answers no. We'll jump-start the first iteration by setting the `bool` object `more` to `true`.

```
#include <iostream>
using namespace std;

extern bool fibon_elem( int, int& );
int main()
{
   int pos, elem;
   char ch;
   bool more = true;

   while ( more )
   {
       cout << "Please enter a position: ";
       cin  >> pos;

       if ( fibon_elem( pos, elem ))
            cout << "element # " << pos
                 << " is " << elem << endl;
       else
            cout << "Sorry. Could not calculate element # "
                 << pos << endl;
```

```
        cout << "would you like to try again? (y/n) ";
        cin >> ch;
        if ( ch != 'y' || ch != 'Y' )
             more = false;
    }
}
```

When compiled and executed, the program generates the following output (my input is highlighted in bold):

```
Please enter a position: 4
element # 4 is 3
would you like to try again? (y/n) y
Please enter a position: 8
element # 8 is 21
would you like to try again? (y/n) y
Please enter a position: 12
element # 12 is 144
would you like to try again? (y/n) n
```

Exercise 2.2

The formula for the Pentagonal numeric sequence is `Pn=n*(3n-1)/2`. This yields the sequence 1, 5, 12, 22, 35, and so on. Define a function to fill a vector of elements passed in to the function calculated to some user-specified position. Be sure to verify that the position specified is valid. Write a second function that, given a vector, displays its elements. It should take a second parameter identifying the type of numeric series the vector represents. Write a `main()` function to exercise these functions.

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;

bool calc_elements( vector<int> &vec, int pos );
void display_elems( vector<int> &vec,
                    const string &title, ostream &os=cout );

int main()
{
    vector<int> pent;
    const string title( "Pentagonal Numeric Series" );

    if ( calc_elements( pent, 0 ))
         display_elems( pent, title );

    if ( calc_elements( pent, 8 ))
```

```
            display_elems( pent, title );

        if ( calc_elements( pent, 14 ))
            display_elems( pent, title );

        if ( calc_elements( pent, 138 ))
            display_elems( pent, title );
    }

    bool calc_elements( vector<int> &vec, int pos )
    {
       if ( pos <= 0 || pos > 64 ){
            cerr << "Sorry. Invalid position: " << pos << endl;
            return false;
       }
       for ( int ix = vec.size()+1; ix <= pos; ++ix )
             vec.push_back( (ix*(3*ix-1))/2 );

       return true;
    }

    void display_elems( vector<int> &vec,
                        const string &title, ostream &os )
    {
       os << '\n' << title << "\n\t";
       for ( int ix = 0; ix < vec.size(); ++ix )
             os << vec[ ix ] << ' ';
       os << endl;
    }
```

When compiled and executed, this program generates the following output:

```
Sorry. Invalid position: 0

Pentagonal Numeric Series
        1 5 12 22 35 51 70 92

Pentagonal Numeric Series
        1 5 12 22 35 51 70 92 117 145 176 210 247 287
Sorry. Invalid position: 138
```

Exercise 2.3

Separate the function to calculate the Pentagonal numeric sequence implemented in Exercise 2.2 into two functions. One function should be inline; it checks the validity of the position. A valid position not as yet calculated causes the function to invoke a second function that does the actual calculation.

I factored `calc_elements()` into the inline `calc_elems()` that if necessary calls the second function, `really_calc_elems()`. To test this reimplementation, I substituted the call of `calc_elements()` in the Exercise 2.2 function with that of `calc_elems()`.

```
extern void really_calc_elems( vector<int>&, int );
inline bool calc_elems( vector<int> &vec, int pos )
{
   if ( pos <= 0 || pos > 64 ){
        cerr << "Sorry. Invalid position: " << pos << endl;
        return false;
   }

   if ( vec.size() < pos )
        really_calc_elems( vec, pos );
   return true;
}
void really_calc_elems( vector<int> &vec, int pos )
{
   for ( int ix = vec.size()+1; ix <= pos; ++ix )
         vec.push_back( (ix*(3*ix-1))/2 );
}
```

Exercise 2.4

Introduce a static local vector to hold the elements of your Pentagonal series. This function returns a const pointer to the vector. It accepts a position by which to grow the vector if the vector is not that size as yet. Implement a second function that, given a position, returns the element at that position. Write a `main()` function to exercise these functions.

```
#include <vector>
#include <iostream>
using namespace std;

inline bool check_validity( int pos )
    {  return ( pos <= 0 || pos > 64 ) ? false : true; }

bool pentagonal_elem( int pos, int &elem )
{
    if ( ! check_validity( pos )){
       cout << "Sorry. Invalid position: " << pos << endl;
       elem = 0;
       return false;
    }
    const vector<int> *pent = pentagonal_series( pos );
    elem = (*pent)[pos-1];
    return true;
}
```

```
const vector<int>*
pentagonal_series( int pos )
{
    static vector<int> _elems;
    if ( check_validity( pos ) && ( pos > _elems.size() ))
         for ( int ix = _elems.size()+1; ix <= pos; ++ix )
               _elems.push_back( (ix*(3*ix-1))/2 );
    return &_elems;
}

int main(){
    int elem;
    if ( pentagonal_elem( 8, elem ))
         cout << "element 8 is " << elem << '\n';
    if ( pentagonal_elem( 88, elem ))
         cout << "element 88 is " << elem << '\n';
    if ( pentagonal_elem( 12, elem ))
         cout << "element 12 is " << elem << '\n';

    if ( pentagonal_elem( 64, elem ))
         cout << "element 64 is " << elem << '\n';
 }
```

When compiled and executed, this program generates the following:

```
element 8 is 92
Sorry. Invalid position: 88
element 12 is 210
element 64 is 6112
```

Exercise 2.5

Implement an overloaded set of `max()` functions to accept (a) two integers, (b) two floats, (c) two strings, (d) a vector of integers, (e) a vector of floats, (f) a vector of strings, (g) an array of integers and an integer indicating the size of the array, (h) an array of floats and an integer indicating the size of the array, and (i) an array of strings and an integer indicating the size of the array. Again, write a `main()` function to exercise these functions.

```
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

inline int max( int t1, int t2 )
       { return t1 > t2 ? t1 : t2; }
```

```
inline float max( float t1, float t2 )
       { return t1 > t2 ? t1 : t2; }

inline string max( const string& t1, const string& t2 )
       { return t1 > t2 ? t1 : t2; }

inline int max( const vector<int> &vec )
       { return *max_element( vec.begin(), vec.end() ); }

inline float max( const vector<float> &vec )
       { return *max_element( vec.begin(), vec.end() ); }

inline string max( const vector<string> &vec )
       { return *max_element( vec.begin(), vec.end() ); }

inline int max( const int *parray, int size )
       { return *max_element( parray, parray+size ); }
inline float max( const float *parray, int size )
       { return *max_element( parray, parray+size ); }

inline string max( const string *parray, int size )
       { return *max_element( parray, parray+size ); }

int main() {
    string sarray[]={ "we", "were", "her", "pride", "of", "ten" };
    vector<string> svec( sarray, sarray+6 );

    int iarray[]={ 12, 70, 2, 169, 1, 5, 29 };
    vector<int> ivec( iarray, iarray+7 );

    float farray[]={ 2.5, 24.8, 18.7, 4.1, 23.9 };
    vector<float> fvec( farray, farray+5 );

    int imax = max( max( ivec ), max( iarray, 7 ));
    float fmax = max( max( fvec ), max( farray, 5 ));
    string smax = max( max( svec ), max( sarray, 6 ));

    cout << "imax should be 169  -- found: " << imax << '\n'
         << "fmax should be 24.8 -- found: " << fmax << '\n'
         << "smax should be were -- found: " << smax << '\n';
}
```

When compiled and executed, this program generates the following:

```
imax should be 169  -- found: 169
fmax should be 24.8 -- found: 24.8
smax should be were -- found: were
```

Exercise 2.6

Reimplement the functions of Exercise 2.5 using templates. Modify the `main()` function accordingly.

The nine nontemplate `max()` functions are replaced by three `max()` function templates. `main()` does not require any changes.

```
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

template <typename Type>
inline Type max( Type t1, Type t2 ){ return t1 > t2 ? t1 : t2; }

template <typename elemType>
inline elemType max( const vector<elemType> &vec )
       { return *max_element( vec.begin(), vec.end() ); }
template <typename arrayType>
inline arrayType max( const arrayType *parray, int size )
       { return *max_element( parray, parray+size ); }

// note: no changes required of main()!
int main() {
    // same as in exercise 2.4
}
```

When compiled and executed, this program generates the same output as the program in Exercise 2.5.

Exercise 3.1

Write a program to read a text file. Store each word in a map. The key value of the map is the count of the number of times the word appears in the text. Define a word exclusion set containing words such as *a, an, or, the, and,* and *but*. Before entering a word in the map, make sure it is not present in the word exclusion set. Display the list of words and their associated count when the reading of the text is complete. As an extension, before displaying the text, allow the user to query the text for the presence of a word.

```
#include <map>
#include <set>
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

void initialize_exclusion_set( set<string>& );
void process_file( map<string,int>&, const set<string>&, ifstream& );
void user_query( const map<string,int>& );
void display_word_count( const map<string,int>&, ofstream& );
```

```
int main()
{
    ifstream ifile( "C:\\My Documents\\column.txt" );
    ofstream ofile( "C:\\My Documents\\column.map" );
    if ( !  ifile || ! ofile ){
         cerr << "Unable to open file -- bailing out!\n";
         return -1;
    }

    set<string> exclude_set;
    initialize_exclusion_set( exclude_set );

    map<string,int> word_count;
    process_file( word_count, exclude_set, ifile );
    user_query( word_count );
    display_word_count( word_count, ofile );
}
void initialize_exclusion_set( set<string> &exs ){
     static string _excluded_words[25] = {
          "the","and","but","that","then","are","been",
          "can","a","could","did","for", "of",
          "had","have","him","his","her","its","is",
          "were","which","when","with","would"
     };

    exs.insert( _excluded_words, _excluded_words+25 );
}

void process_file( map<string,int> &word_count,
                   const set<string> &exclude_set, ifstream &ifile )
{
    string word;
    while ( ifile >> word )
    {
            if ( exclude_set.count( word ))
                 continue;
            word_count[ word ]++;
    }
}

void user_query( const map<string,int> &word_map )
{
    string search_word;
    cout << "Please enter a word to search: q to quit";
    cin >> search_word;
    while ( search_word.size() && search_word != "q" )
    {
            map<string,int>::const_iterator it;
```

```
            if (( it = word_map.find( search_word ))
                    != word_map.end() )
                  cout << "Found! "  << it->first
                       << " occurs " << it->second
                       << " times.\n";
             else cout << search_word
                       << " was not found in text.\n";
              cout << "\nAnother search? (q to quit) ";
              cin >> search_word;
      }
}

void
display_word_count( const map<string,int> &word_map, ofstream &os )
{
    map<string,int>::const_iterator
                iter = word_map.begin(),
                end_it = word_map.end();
    while ( iter != end_it ){
            os << iter->first  << " ( "
               << iter->second << " )" << endl;
           ++iter;
    }
    os << endl;
}
```

Here is a small piece of text processed by the program. I removed the punctuation from the text because our program does not handle punctuation:

```
MooCat is a long-haired white kitten with large
black patches Like a cow looks only he is a kitty
poor kitty Alice says cradling MooCat in her arms
pretending he is not struggling to break free
```

Here is a snapshot of the interactive session initiated by `user_query()`. Notice that although *a* occurs two times the text, it is an entry in the excluded word set and is not entered in the map of words found in the text.

```
Please enter a word to search: q to quit Alice
Found! Alice occurs 1 times.

Another search? (q to quit) MooCat
Found! MooCat occurs 2 times.

Another search? (q to quit) a
a was not found in text.

Another search? (q to quit) q
```

Exercise 3.2

Read in a text file — it can be the same one as in Exercise 3.1 — storing it in a vector. Sort the vector by the length of the string. Define a function object to pass to `sort()`; it should accept two strings and return `true` if the first string is shorter than the second. Print the sorted vector.

Let's begin by defining the function object to pass to `sort()`:

```
class LessThan {
public:
   bool operator()( const string & s1,
                    const string & s2 )
      { return s1.size() < s2.size(); }
};
```

The call to sort looks like this:

```
sort( text.begin(), text.end(), LessThan() );
```

The main program looks like this:

```
int main()
{
    ifstream ifile( "C:\\My Documents\\MooCat.txt" );
    ofstream ofile( "C:\\My Documents\\MooCat.sort" );

    if ( !  ifile || ! ofile ){
         cerr << "Unable to open file -- bailing out!\n";
         return -1;
    }

    vector<string> text;
    string word;

    while ( ifile >> word )
            text.push_back( word );

    sort( text.begin(), text.end(), LessThan() );
    display_vector( text, ofile );
}
```

`display_vector()` is a function template parameterized on the element type of the vector passed to it to display:

```
template <typename elemType>
void display_vector( const vector<elemType> &vec,
                     ostream &os=cout, int len= 8 )
{
    vector<elemType>::const_iterator
             iter = vec.begin(),
             end_it = vec.end();
```

```
    int elem_cnt = 1;
    while ( iter != end_it )
          os << *iter++
                << ( !( elem_cnt++ % len ) ? '\n' : ' ');
    os << endl;
}
```

The input file is the same text used in Exercise 3.1. The output of this program looks like this:

```
a a a is to in is he
is he not cow her says poor only
Like arms with free break Alice kitty kitty
looks black large white MooCat kitten MooCat patches
cradling pretending struggling long-haired
```

If we want to sort the words within each length alphabetically, we would first invoke `sort()` with the default less-than operator and then invoke `stable_sort()`, passing it the LessThan function object. `stable_sort()` maintains the relative order of elements meeting the same sorting criteria.

Exercise 3.3

Define a map for which the index is the family surname and the key is a vector of the children's names. Populate the map with at least six entries. Test it by supporting user queries based on a surname and printing all the map entries.

The map uses a string as an index and a vector of strings for the children's names. This is declared as follows:

```
map< string, vector<string> > families;
```

To simplify the declaration of the map, I've defined a typedef to alias `vstring` as an alternative name for a vector that contains string elements. (You likely wouldn't have thought of this because the typedef mechanism is not introduced until Section 4.6. The typedef mechanism allows us to provide an alternative name for a type. It is generally used to simplify the declaration of a complex type.)

```
#include <map>
typedef vector<string> vstring;
map< string, vstring > families;
```

We get our family information from a file. Each line of the file stores the family name and the name of each child:

```
surname child1 child2 child3 ... childN
```

Reading the file and populating the map are accomplished in `populate_map()`:

```
void populate_map( ifstream &nameFile, map<string,vstring> &families )
{
```

```
    string textline;
    while ( getline( nameFile, textline ))
    {
        string fam_name;
        vector<string> child;
        string::size_type
            pos = 0, prev_pos = 0,
           text_size = textline.size();

        // ok: find each word separated by a space
        while ((  pos = textline.find_first_of( ' ', pos ))
                    != string::npos )
        {
             // figure out end points of the substring
             string::size_type end_pos = pos - prev_pos;

             // if prev_pos not set, this is the family name
             // otherwise, we are reading the children ...
             if ( ! prev_pos )
                  fam_name = textline.substr( prev_pos, end_pos );
             else child.push_back(textline.substr(prev_pos,end_pos));
             prev_pos = ++pos;
        }

        // now handle last child
        if ( prev_pos < text_size )
             child.push_back(textline.substr(prev_pos,pos-prev_pos));

        if ( ! families.count( fam_name ))
             families[ fam_name ] = child;
        else cerr << "Oops! We already have a "
                  << fam_name << " family in our map!\n";
     }
}
```

`getline()` is a standard library function. It reads a line of the file up to the delimiter specified by its third parameter. The default delimiter is the newline character, which is what we use. The line is placed in the ssecond parameter string.

The next portion of the function pulls out, in turn, the family surname and the name of each child. `substr()` is a string class operation that returns a new string object from the substring delimited by the two position arguments. Finally, the family is entered into the map if one is not already present.

To display the map, we define a `display_map()` function. It prints entries in this general format:

```
The lippman family has 2 children: danny anna
```

Here is the `display_map()` implementation:

```
void display_map( const map<string,vstring> &families, ostream &os )
{
   map<string,vstring>::const_iterator
               it = families.begin(),
                end_it = families.end();

   while ( it != end_it )
   {
         os << "The " << it->first << " family ";
         if ( it->second.empty() )
              os << "has no children\n";
         else
         {  // print out vector of children
             os   << "has " << it->second.size()  << " children: ";
            vector<string>::const_iterator
                    iter = it->second.begin(),
                    end_iter = it->second.end();
            while ( iter != end_iter )
                  { os << *iter << " "; ++iter; }
            os << endl;
         }
         ++it;
    }
}
```

We must also allow users to query the map as to the presence of a family. If the family is present, we display the family name and number of children in the same way that `display_map()` does for the entire map. (As an exercise, you might factor out the common code between the two functions.) I've named this function `query_map()`.

```
void query_map( const string &family,
                const map<string,vstring> &families )
{
    map<string,vstring>::const_iterator
         it = families.find( family );

    if ( it == families.end() ){
         cout << "Sorry. The " << family
              << " is not currently entered.\n";
         return;
    }

    cout << "The " << family;
    if ( ! it->second.size() )
         cout << " has no children\n";
    else { // print out vector of children
          cout << " has " << it->second.size() << " children: ";
          vector<string>::const_iterator
                  iter = it->second.begin(),
                  end_iter = it->second.end();
```

```
            while ( iter != end_iter )
                  { cout << *iter << " "; ++iter; }
             cout << endl;
        }
    }
```

The `main()` program is implemented as follows:

```
int main()
{
   map< string, vstring > families;
   ifstream nameFile( "C:\\My Documents\\families.txt" );

   if ( ! nameFile ) {
         cerr << "Unable to find families.txt file. Bailing Out!\n";
         return;
   }
   populate_map( nameFile, families );

   string family_name;
   while ( 1 ){ //!! loop until user says to quit ...
      cout << "Please enter a family name or q to quit ";
      cin >> family_name;
      if ( family_name == "q" )
           break;
      query_map( family_name, families );
   }
   display_map( families );
}
```

The `families.txt` file contains the following six entries:

```
lippman danny anna
smith john henry frieda
mailer tommy june
franz
orlen orley
ranier alphonse lou robert brodie
```

When compiled and executed, the program generates the following results. My responses are highlighted in bold.

```
Please enter a family name or q to quit ranier
The ranier family  has 4 children: alphonse lou robert brodie
Please enter a family name or q to quit franz
The franz family  has no children
Please enter a family name or q to quit kafka
Sorry. The kafka family is not currently entered.
Please enter a family name or q to quit q
The franz family has no children
The lippman family has 2 children: danny anna
The mailer family has 2 children: tommy june
```

```
The orlen family has 1 children: orley
The ranier family has 4 children: alphonse lou robert brodie
The smith family has 3 children: john henry frieda
```

Exercise 3.4

Write a program to read a sequence of integer numbers from standard input using an istream_iterator. Write the odd numbers into one file using an ostream_iterator. Each value should be separated by a space. Write the even numbers into a second file, also using an ostream_iterator. Each of these values should be placed on a separate line.

To read a sequence of integers from standard input, we define two istream_iterators: one bound to `cin`, and the second representing end-of-file.

```
istream_iterator<int> in( cin ), eos;
```

Next, we define a vector to hold the elements read:

```
vector< int > input;
```

To perform the reading, we use the `copy()` generic algorithm:

```
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
   vector< int > input;
   istream_iterator in( cin ), eos;

   copy( in, eos, back_inserter( input ));

   // ...
}
```

The `back_inserter()` is necessary because `copy()` uses the assignment operator to copy each element. Because `input` is empty, the first element assignment would cause an overflow error. `back_inserter()` overrides the assignment operator. The elements are now inserted using `push_back()`.

We partition the elements into even and odd using the `partition()` generic algorithm and an `even_elem` function object that evaluates to `true` if the value is even:

```
class even_elem {
public:
   bool operator()( int elem )
        { return elem%2 ? false : true; }
```

```
};

vector<int>::iterator division =
     partition( input.begin(), input.end(), even_elem() );
```

We need two ostream_iterators: one for the even number file and one for the odd number file. We first open two files for output using the ofstream class:

```
#include <fstream>
ofstream even_file( "C:\\My Documents\\even_file" ),
          odd_file( "C:\\My Documents\\odd_file" );

if ( ! even_file || ! odd_file )
{
     cerr << "arghh!! unable to open the output files. bailing out!";
     return -1;
}
```

We bind our two ostream_iterators to the respective ofstream objects. The second string argument indicates the delimiter to output following the output of each element.

```
ostream_iterator<int> even_iter( even_file, "\n" ),
                      odd_iter( odd_file, " " );
```

Finally, we use the `copy()` generic algorithm to output the partitioned elements:

```
copy( input.begin(), division, even_iter );
copy( division, input.end(), odd_iter );
```

For example, I entered the following sequence of numbers:

```
2 4 5 3 9 5 2 6 8 1 8 4 5 7 3
```

At that point, `even_file` contained the following values: `2 4 4 8 8 6 2`, whereas `odd_file` contained these values: `5 9 1 3 5 5 7 3`. The `partition()` algorithm does not preserve the order of the values. If preserving the order of the values is important, we would instead use the `stable_partition()` generic algorithm. Both the `stable_sort()` and `stable_partition()` algorithms maintain the elements' relative order.

Exercise 4.1

Create a `Stack.h` and a `Stack.suffix`, where `suffix` is whatever convention your compiler or project follows. Write a `main()` function to exercise the full public interface, and compile and execute it. Both the program text file and `main()` must include `Stack.h`:

```
#include "Stack.h"
```

The header file for our Stack class contains the necessary header file inclusions and the actual class declaration:

```
#include <string>
#include <vector>
using namespace std;
```

```
class Stack {
public:
   bool   push( const string& );
   bool   pop (  string &elem );
   bool   peek( string &elem );
   bool   empty() const { return _stack.empty(); }
   bool   full()  const { return _stack.size() == _stack.max_size(); }
   int    size()  const { return _stack.size(); }

private:
   vector<string> _stack;
};
```

The Stack program text file contains the definition of the `push()`, `pop()`, and `peek()` member functions. Under Visual C++, the file is named `Stack.cpp`. It must include the Stack class header file.

```
#include "Stack.h"
bool Stack::pop( string &elem ){
    if ( empty() ) return false;
    elem = _stack.back();
    _stack.pop_back();
    return true;
}

bool Stack::peek( string &elem ){
    if ( empty() ) return false;
    elem = _stack.back();
    return true;
}

bool Stack::push( const string &elem ){
    if ( full() ) return false;
    _stack.push_back( elem );
    return true;
}
```

Here is a small program to exercise the Stack class interface. It reads in a sequence of strings from standard input, pushing each one onto the stack until either end-of-file occurs or the stack is full:

```
int main() {
    Stack st;
    string str;

    while ( cin >> str && ! st.full() )
            st.push( str );

    if ( st.empty() ) {
```

```
            cout << '\n' << "Oops: no strings were read -- bailing out\n ";
             return 0;
        }
        st.peek( str );
        if ( st.size() == 1 && str.empty() ) {
            cout << '\n' << "Oops: no strings were read -- bailing out\n ";
             return 0;
        }
        cout << '\n' << "Read in " << st.size() << " strings!\n"
             << "The strings, in reverse order: \n";

        while ( st.size() )
           if ( st.pop( str ))
                cout << str << ' ';

      cout << '\n' << "There are now " << st.size()
           << " elements in the stack!\n";
}
```

To test the program, I typed in the last sentence of the James Joyce novel, *Finnegans Wake*. The following is the output generated by the program (my input is in bold):

```
A way a lone a last a loved a long the
Read in 11 strings!
The strings, in reverse order:
the long a loved a last a lone a way A
There are now 0 elements in the stack!
```

Exercise 4.2

Extend the Stack class to support both a `find()` and a `count()` operation. `find()` returns `true` or `false` depending on whether the value is found. `count()` returns the number of occurrences of the string. Reimplement the `main()` of Exercise 4.1 to invoke both functions.

We implement these two functions simply by using the corresponding generic algorithms of the same names:

```
#include <algorithm>
bool Stack::find( const string &elem ) const {
   vector<string>::const_iterator end_it = _stack.end();
   return ::find( _stack.begin(), end_it, elem ) != end_it;
}

int Stack::count( const string &elem ) const
    { return ::count( _stack.begin(), _stack.end(), elem ); }
```

The global scope operator is necessary for the invocation of the two generic algorithms. Without the global scope operator, for example, the unqualified invocation of

find() within find() recursively invokes the member instance of find()! The Stack class declaration is extended to include the declarations of these two functions:

```
class Stack {
public:
   bool   find(  const string &elem ) const;
   int    count( const string &elem ) const;

   // ... everything else the same ...
};
```

The program now inquires of the user which word she would like to search for and reports whether it is within the stack and, if so, how many times it occurs:

```
int main()
{
    Stack st;
    string str;
    while ( cin >> str && ! st.full() )
            st.push( str );
    // check for empty stack as before ...

    cout << '\n' << "Read in " << st.size() << " strings!\n";
    cin.clear(); // clear end-of-file set ...

    cout << "what word to search for? ";
    cin  >> str;

    bool found = st.find( str );
    int  count = found ? st.count( str ) : 0;

    cout << str << (found ? " is " : " isn\'t " ) << "in the stack. ";
    if ( found )
         cout << "It occurs " << count << " times\n";
}
```

Here is an interactive execution of the program. The items highlighted in bold are what I entered:

```
A way a lone a last a loved a long the
Read in 11 strings!
what word to search for? a
a is in the stack. It occurs 4 times
```

Exercise 4.3

Consider the following global data:

```
string program_name;
string version_stamp;
int version_number;
int tests_run;
```

```
int tests_passed;
```

Write a class to wrap around this data.

Why might we wish to do this? By wrapping these global objects within a class, we encapsulate their direct access within a small set of functions. Moreover, the names of the objects are now hidden behind the scope of the class and cannot clash with other global entities. Because we wish only a single instance of each global object, we declare each one to be a static class member as well as the member functions that access them.

```
#include <string>
using std::string;

class globalWrapper {
public:
    static int tests_passed()     { return _tests_passed; }
    static int tests_run()        { return _tests_run; }
    static int version_number()   { return _version_number; }
    static string version_stamp() { return _version_stamp; }
    static string program_name()  { return _program_name; }

    static void tests_passed( int nval )  { _tests_passed = nval; }
    static void tests_run( int nval )     { _tests_run = nval; }

    static void version_number( int nval )
                { _version_number = nval; }

    static void version_stamp( const string& nstamp )
                { _version_stamp = nstamp; }

    static void program_name( const string& npn )
                { _program_name = npn; }

private:
   static string  _program_name;
   static string  _version_stamp;
   static int     _version_number;
   static int     _tests_run;
   static int     _tests_passed;
};

string globalWrapper::_program_name;
string globalWrapper::_version_stamp;
int globalWrapper::_version_number;
int globalWrapper::_tests_run;
int globalWrapper::_tests_passed;
```

Exercise 4.4

A user profile consists of a login, the actual user name, the number of times logged on, the number of guesses made, the number of correct guesses, the current level — one of beginner, intermediate, advanced, guru — and the percentage correct (this latter may be computed or stored). Provide a UserProfile class. Support input and output, equality and inequality. The constructors should allow for a default user level and default login name of "guest." How might you guarantee that each guest login for a particular session is unique?

```
class UserProfile {
public:
    enum uLevel { Beginner, Intermediate, Advanced, Guru };

    UserProfile( string login, uLevel = Beginner );
    UserProfile();

    // default memberwise initialization and copy sufficient
    // no explicit copy constructor or copy assignment operator
    // no destructor necessary ...

    bool operator==( const UserProfile& );
    bool operator!=( const UserProfile &rhs );

    // read access functions
    string login() const { return _login; }
    string user_name() const { return _user_name; }
    int login_count() const { return _times_logged; }
    int guess_count() const { return _guesses; }
    int guess_correct() const { return _correct_guesses; }
    double guess_average() const;
    string level() const;

    // write access functions
    void reset_login( const string &val ){ _login = val; }
    void user_name( const string &val ){ _user_name = val; }

    void reset_level( const string& );
    void reset_level( uLevel newlevel ) { _user_level = newlevel; }

    void reset_login_count( int val ){ _times_logged = val; }
    void reset_guess_count( int val ){ _guesses = val; }
    void reset_guess_correct( int val ){ _correct_guesses = val; }

    void bump_login_count( int cnt=1 ){ _times_logged += cnt; }
    void bump_guess_count( int cnt=1 ){ _guesses += cnt; }
    void bump_guess_correct(int cnt=1){ _correct_guesses += cnt;}
```

```
 private:
    string _login;
    string _user_name;
    int    _times_logged;
    int    _guesses;
    int    _correct_guesses;
    uLevel _user_level;

    static map<string,uLevel> _level_map;
    static void init_level_map();
    static string guest_login();
};

inline double UserProfile::guess_average() const
{
   return _guesses
         ? double(_correct_guesses) / double(_guesses) * 100
         : 0.0;
}
inline UserProfile::UserProfile( string login, uLevel level )
    : _login( login ),   _user_level( level ),
      _times_logged( 1 ), _guesses( 0 ), _correct_guesses( 0 ){}

#include <cstdlib>

inline UserProfile::UserProfile()
    : _login( "guest" ), _user_level( Beginner ),
      _times_logged( 1 ), _guesses( 0 ), _correct_guesses( 0 )
{
   static int id = 0;
   char buffer[ 16 ];

   // _itoa() is a Standard C library function
   // turns an integer into an ascii representation
      _itoa( id++, buffer, 10 );

   // add a unique id during session to guest login
   _login += buffer;
}

inline bool UserProfile::
operator==( const UserProfile &rhs )
{
     if ( _login == rhs._login &&
          _user_name == rhs._user_name )
               return true;
     return false;
}
```

```
inline bool UserProfile::
operator !=( const UserProfile &rhs ){ return ! ( *this == rhs ); }

inline string UserProfile::level() const {
    static string _level_table[] = {
           "Beginner", "Intermediate", "Advanced", "Guru" };
    return _level_table[ _user_level ];
}

ostream& operator<<( ostream &os, const UserProfile &rhs )
{ // output of the form: stanl Beginner 12 100 10 10%
     os << rhs.login() << ' '
        << rhs.level() << ' '
        << rhs.login_count() << ' '
        << rhs.guess_count() << ' '
        << rhs.guess_correct() << ' '
        << rhs.guess_average() << endl;
   return os;
}
// overkill ... but it allows a demonstration ...
map<string,UserProfile::uLevel> UserProfile::_level_map;

void UserProfile::init_level_map(){
    _level_map[ "Beginner" ] = Beginner;
    _level_map[ "Intermediate" ] = Intermediate;
    _level_map[ "Advanced" ] = Advanced;
    _level_map[ "Guru" ] = Guru;
}

inline void UserProfile::reset_level( const string &level ){
     map<string,uLevel>::iterator it;
     if ( _level_map.empty() )
          init_level_map();

     // confirm level is a recognized user level ...
     _user_level =
         (( it = _level_map.find( level )) != _level_map.end() )
               ? it->second : Beginner;
}

istream& operator>>( istream &is, UserProfile &rhs )
{   // yes, this assumes the input is valid ...
    string login, level;
    is >> login >> level;

    int lcount, gcount, gcorrect;
    is >> lcount >> gcount >> gcorrect;
    rhs.reset_login( login );
```

```
        rhs.reset_level( level );

        rhs.reset_login_count( lcount );
        rhs.reset_guess_count( gcount );
        rhs.reset_guess_correct( gcorrect );

        return is;
    }
```

Here is a small program to exercise our UserProfile class:

```
int main()
{
   UserProfile anon;
   cout << anon; // test out output operator

   UserProfile anon_too; // to see if we get unique id
   cout << anon_too;

   UserProfile anna( "AnnaL", UserProfile::Guru );
   cout << anna;
   anna.bump_guess_count( 27 );
   anna.bump_guess_correct( 25 );
   anna.bump_login_count();
   cout << anna;

   cin >> anon; // test out input operator
   cout << anon;
}
```

When compiled and executed, this program generates the following output (my responses are highlighted in bold):

```
guest0 Beginner 1 0 0 0
guest1 Beginner 1 0 0 0
AnnaL Guru 1 0 0 0
AnnaL Guru 2 27 25 92.5926
robin Intermediate 1 8 3
robin Intermediate 1 8 3 37.5
```

Exercise 4.5

Implement a 4x4 Matrix class supporting at least the following general interface: addition and multiplication of two Matrix objects, a `print()` member function, a compound += operator, and subscripting supported through a pair of overloaded function call operators, as follows:

```
float& operator()( int row, int column );
float  operator()( int row, int column ) const;
```

Provide a default constructor taking an optional 16 data values and a constructor taking

an array of 16 elements. You do not need a copy constructor, copy assignment operator, or destructor for this class (these are required in Chapter 6 when we reimplement the Matrix class to support arbitrary rows and columns).

```
#include <iostream>
typedef float elemType; // for transition into a template

class Matrix
{
    // friends are not affected by the access level they are
    // declared in. I like to place them at class beginning
    friend Matrix operator+( const Matrix&, const Matrix& );
    friend Matrix operator*( const Matrix&, const Matrix& );

public:
    Matrix( const elemType* );
    Matrix( elemType=0.,elemType=0.,elemType=0.,elemType=0.,
            elemType=0.,elemType=0.,elemType=0.,elemType=0.,
            elemType=0.,elemType=0.,elemType=0.,elemType=0.,
            elemType=0.,elemType=0.,elemType=0.,elemType=0. );
    // don't need copy constructor, destructor,
    // or copy assignment operator for the Matrix class

    // simplifies transition to general matrix
    int rows() const { return 4; }
    int cols() const { return 4; }

    ostream& print( ostream& ) const;
    void operator+=( const Matrix& );

    elemType operator()( int row, int column ) const
           { return _matrix[ row ][ column ]; }

    elemType& operator()( int row, int column )
           { return _matrix[ row ][ column ]; }
private:
    elemType _matrix[4][4];
};

inline ostream& operator<<( ostream& os, const Matrix &m )
       { return m.print( os ); }

Matrix operator+( const Matrix &m1, const Matrix &m2 ){
    Matrix result( m1 );
    result += m2;
    return result;
}
Matrix operator*( const Matrix &m1, const Matrix &m2 ){
    Matrix result;
```

```
    for ( int ix = 0; ix < m1.rows(); ix++ )
        for ( int jx = 0; jx < m1.cols(); jx++ ){
               result( ix, jx ) = 0;
               for ( int kx = 0; kx < m1.cols(); kx++ )
                     result( ix, jx ) += m1( ix, kx ) * m2( kx, jx );
             }
    return result;
}

void Matrix::operator+=( const Matrix &m ){
    for ( int ix = 0; ix < 4; ++ix )
        for ( int jx = 0; jx < 4; ++jx )
            _matrix[ix][jx] += m._matrix[ix][jx];
}

ostream& Matrix::print( ostream &os ) const {
    int cnt = 0;
    for ( int ix = 0; ix < 4; ++ix )
        for ( int jx = 0; jx < 4; ++jx, ++cnt ){
            if ( cnt &&  !( cnt % 8 )) os << endl;
            os << _matrix[ix][jx] << ' ';
        }
    os << endl;
    return os;
}

Matrix::Matrix( const elemType *array ){
    int array_index = 0;
    for ( int ix = 0; ix < 4; ++ix )
          for ( int jx = 0; jx < 4; ++jx )
                _matrix[ix][jx] = array[array_index++];
}

Matrix::Matrix(
        elemType a11, elemType a12, elemType a13, elemType a14,
        elemType a21, elemType a22, elemType a23, elemType a24,
        elemType a31, elemType a32, elemType a33, elemType a34,
        elemType a41, elemType a42, elemType a43, elemType a44 )
{
   _matrix[0][0] = a11; _matrix[0][1] = a12;
   _matrix[0][2] = a13; _matrix[0][3] = a14;
   _matrix[1][0] = a21; _matrix[1][1] = a22;
   _matrix[1][2] = a23; _matrix[1][3] = a24;
   _matrix[2][0] = a31; _matrix[2][1] = a32;
   _matrix[2][2] = a33; _matrix[2][3] = a34;
   _matrix[3][0] = a41; _matrix[3][1] = a42;
   _matrix[3][2] = a43; _matrix[3][3] = a44;
}
```

Here is a small program that exercises a portion of the Matrix class interface:

```
int main()
{
   Matrix m;
   cout << m << endl;

   elemType ar[16]={
      1., 0., 0., 0., 0., 1., 0., 0.,
      0., 0., 1., 0., 0., 0., 0., 1. };

   Matrix identity( ar );
   cout << identity << endl;

   Matrix m2( identity );
   m = identity;
   cout << m2 << endl; cout << m  << endl;

   elemType ar2[16] = {
      1.3, 0.4, 2.6, 8.2, 6.2, 1.7, 1.3, 8.3,
      4.2, 7.4, 2.7, 1.9, 6.3, 8.1, 5.6, 6.6 };
   Matrix m3( ar2 ); cout << m3 << endl;
   Matrix m4 = m3 * identity; cout << m4 << endl;
   Matrix m5 = m3 + m4; cout << m5 << endl;
   m3 += m4; cout << m3 << endl;
}
```

When compiled and executed, this program generates the following output:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

1 0 0 0 0 1 0 0
0 0 1 0 0 0 0 1

1 0 0 0 0 1 0 0
0 0 1 0 0 0 0 1

1 0 0 0 0 1 0 0
0 0 1 0 0 0 0 1

1.3 0.4 2.6 8.2 6.2 1.7 1.3 8.3
4.2 7.4 2.7 1.9 6.3 8.1 5.6 6.6
1.3 0.4 2.6 8.2 6.2 1.7 1.3 8.3
4.2 7.4 2.7 1.9 6.3 8.1 5.6 6.6

2.6 0.8 5.2 16.4 12.4 3.4 2.6 16.6
8.4 14.8 5.4 3.8 12.6 16.2 11.2 13.2

2.6 0.8 5.2 16.4 12.4 3.4 2.6 16.6
8.4 14.8 5.4 3.8 12.6 16.2 11.2 13.2
```

Exercise 5.1

Implement a two-level stack hierarchy. The base class is a pure abstract Stack class that minimally supports the following interface: `pop()`, `push()`, `size()`, `empty()`, `full()`, `peek()`, and `print()`. The derived classes are LIFO_Stack and Peekback_Stack. The Peekback_Stack allows the user to retrieve the value of any element in the stack without modifying the stack itself.

The two derived classes implement the actual element container using a vector. To display the vector, I use a `const_reverse_iterator`. This supports traversing the vector from the back to the front.

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

typedef string elemType;
class Stack {
public:
    virtual ~Stack(){}
    virtual bool pop( elemType& ) = 0;
    virtual bool push( const elemType& ) = 0;
    virtual bool peek( int index, elemType& ) = 0;

    virtual int  top() const = 0;
    virtual int  size() const = 0;

    virtual bool empty() const = 0;
    virtual bool full() const = 0;
    virtual void print( ostream& =cout ) const = 0;
};

ostream& operator<<( ostream &os, const Stack &rhs )
      { rhs.print(); return os; }

class LIFO_Stack : public Stack {
public:
   LIFO_Stack( int capacity = 0 ) : _top( 0 )
        { if ( capacity ) _stack.reserve( capacity ); }
   int  size()  const { return _stack.size(); }
   bool empty() const { return ! _top; }
   bool full()  const { return size() >= _stack.max_size(); }
   int  top()   const { return _top; }
   void print( ostream &os=cout ) const;

   bool pop(  elemType &elem );
   bool push( const elemType &elem );
```

```
    bool peek( int, elemType& ){ return false; }
private:
    vector< elemType > _stack;
    int _top;
};

bool LIFO_Stack::pop( elemType &elem ){
     if ( empty() ) return false;
     elem = _stack[ --_top ];
     _stack.pop_back();
     return true;
}

bool LIFO_Stack::push( const elemType &elem ){
     if ( full() ) return false;
     _stack.push_back( elem );
     ++_top;
     return true;
}
void LIFO_Stack::print( ostream &os=cout ) const {
    vector<elemType>::const_reverse_iterator
           rit = _stack.rbegin(),
           rend = _stack.rend();

    os << "\n\t";
    while ( rit != rend )
            os << *rit++ << "\n\t";

    os << endl;
}
```

The implementation of the Peekback_Stack duplicates that of LIFO_Stack except for the implementation of `peek()`:

```
bool Peekback_Stack::
peek( int index, elemType &elem )
{
    if ( empty() )
         return false;

    if ( index < 0 || index >= size() )
         return false;

    elem = _stack[ index ];
    return true;
}
```

Here is a small program that exercises the inheritance hierarchy. The nonmember `peek()` instance accepts an abstract Stack reference and invokes the virtual `peek()` member function, which is unique to each derived class.

```
void peek( Stack &st, int index )
{
   cout  << endl;
   string t;
   if ( st.peek( index, t ))
        cout << "peek: " << t;
   else cout << "peek failed!";
   cout << endl;
}

int main()
{
   LIFO_Stack st;
   string str;
   while ( cin >> str && ! st.full() )
           st.push( str );

   cout << '\n' << "About to call peek() with LIFO_Stack" << endl;
   peek( st, st.top()-1 );
   cout << st;

   Peekback_Stack pst;

   while ( ! st.empty() ){
      string t;
      if ( st.pop( t ))
           pst.push( t );
   }

   cout << "About to call peek() with Peekback_Stack" << endl;
   peek( pst, pst.top()-1 );
   cout << pst;
}
```

When compiled and executed, this program generates the following output :

```
once upon a time
About to call peek() with LIFO_Stack
peek failed!

     time
     a
     upon
     once

 About to call peek() with Peekback_Stack
 peek: once

     once
     upon
```

```
a
time
```

Exercise 5.2

Reimplement the class hierarchy of Exercise 5.1 so that the base Stack class implements the shared, type-independent members.

The reimplementation of the class hierarchy illustrates a concrete class hierarchy; that is, we replace our pure abstract Stack class with our implementation of LIFO_Stack, renaming it Stack. Although Stack serves as a base class, it also represents actual objects within our applications. Thus it is termed a concrete base class. Peekback_Stack is derived from Stack. In this implementation, it inherits all the members of Stack except `peek()`, which it overrides. Only the `peek()` member function and the destructor of Stack are virtual. The definitions of the member functions are the same and are not shown.

```
class Stack {
public:
    Stack( int capacity = 0 ): _top( 0 )
    {
       if ( capacity )
            _stack.reserve( capacity );
    }
    virtual ~Stack(){}

    bool pop( elemType& );
    bool push( const elemType& );
    virtual bool peek( int, elemType& )
           { return false; }

    int  size()  const { return _stack.size(); }
    int  top()   const { return _top; }

    bool empty() const { return ! _top; }
    bool full()  const { return size() >= _stack.max_size(); }
    void print( ostream&=cout );

protected:
    vector<elemType> _stack;
    int _top;
};

class Peekback_Stack : public Stack {
public:
   Peekback_Stack( int capacity = 0 )
       : Stack( capacity ) {}
```

```
    bool peek( int index, elemType &elem );
};
```

Exercise 5.3

A type/subtype inheritance relationship in general reflects an *is-a* relationship: A range-checking ArrayRC is a kind of Array, a Book is a kind of LibraryRentalMaterial, an AudioBook is a kind of Book, and so on. Which of the following pairs reflects an is-a relationship?

(a) member function isA_kindOf function

This reflects an is-a relationship. A member function is a specialized instance of a function. Both have a return type, a name, a parameter list, and a definition. In addition, a member function belongs to a particular class, may or may not be a virtual, const, or static member function, and so on. Inheritance correctly models the relationship.

(b) member function isA_kindOf class

This does not reflect an is-a relationship. A member function has a class that it is a member of, but it is not a specialized instance of a class. Inheritance incorrectly models the relationship.

(c) constructor isA_kindOf member function

This reflects an is-a relationship. A constructor is a specialized instance of a member function. A constructor must be a member function; however, it has specialized characteristics. Inheritance correctly models the relationship.

(d) airplane isA_kindOf vehicle

This reflects an is-a relationship. An airplane is a kind of vehicle. A vehicle is an abstract class. Airplane is also an abstract class and subsequently is likely to be inherited from. Inheritance correctly models the relationship.

(e) motor isA_kindOf truck

This does not reflect an is-a relationship. A motor is part of a truck. A truck has a motor. Inheritance incorrectly models the relationship.

(f) circle isA_kindOf geometry

This reflects an is-a relationship. A circle is a specialized instance of geometry — of 2D geometry. Geometry is an abstract base class. Circle is a concrete specialization of geometry. Inheritance correctly models the relationship.

(g) square isA_kindOf rectangle

This reflects an is-a relationship. Like a circle, a rectangle is a specialized instance of geometry. A square is a further specialization — a rectangle in which each side is equal. Inheritance correctly models the relationship.

```
(h) automobile isA_kindOf airplane
```

This is neither a has-a nor an is-a relationship. Both an automobile and an airplane are kinds of vehicles. Inheritance incorrectly models the relationship.

```
(i) borrower isA_kindOf library
```

This does not reflect an is-a relationship. A borrower is an object (or component) of a library. A library has one or more borrowers. A borrower is a member of a library but is not a kind of library! Inheritance incorrectly models the relationship.

Exercise 5.4

A library supports the following categories of lending materials, each with its own check-out and check-in policy. Organize these into an inheritance hierarchy:

```
book          audio book
record        children's puppet
video         Sega video game
rental book   Sony Playstation video game
CD-ROM book   Nintendo video game
```

An inheritance hierarchy moves from the most abstract to the most specific. In this example, we are given concrete instances of materials loaned by a library. Our task is twofold. First, we must group common abstractions: the four book abstractions and the three video game abstractions. Second, we must provide additional classes that can serve as an abstract interface for the concrete instances. This must be done at two levels: at the level of each family of concrete classes, such as our books, and at the level of the entire library lending hierarchy.

For example, Sega, Sony Playstation, and Nintendo video games are specific instances of video games. To tie them together, I've introduced an abstract video game class. I've designated the book class as a concrete base class with the three other kinds of books as specialized instances. Finally, we need a root base class of our library lending materials.

In the following hierarchy, each tab represents an inheritance relationship. For example, audio, rental, and CD-ROM are inherited from book, which in turn is inherited from the abstract library_lending_material.

```
library_lending_material
   book
      audio book
      rental book
      CD-ROM book
   children's puppet
   record
   video
   video game
```

```
Sega
Sony Playstation
Nintendo
```

Exercise 6.1

Rewrite the following class definition to make it a class template:

```
class example {
public:
      example( double min, double max );
      example( const double *array, int size );

      double& operator[]( int index );
      bool operator==( const example1& ) const;

      bool insert( const double*, int );
      bool insert( double );

      double min() const { return _min; }
      double max() const { return _max; }

      void min( double );
      void max( double );

      int count( double value ) const;
private:
      int     _size;
      double *_parray;
      double  _min;
      double  _max;
};
```

To transform the example class into a template, we must identify and factor out each dependent data type. `_size`, for example, is of type `int`. Might that vary with different user-specified instances of example? No. `_size` is an invariant data member that holds a count of the elements addressed by `_parray`. `_parray`, however, may address elements of varying types: `int`, `double`, `float`, string, and so on. We want to parameterize the data type of the members `_parray`, `_min`, and `_max`, as well as the return type and signature of some of the member functions.

```
template <typename elemType>
class example {
public:
      example( const elemType &min, const elemType &max );
      example( const elemType *array, int size );

      elemType& operator[]( int index );
      bool operator==( const example1& ) const;
```

```
        bool insert( const elemType*, int );
        bool insert( const elemType& );

        elemType min() const { return _min; };
        elemType max() const { return _max; };

        void min( const elemType& );
        void max( const elemType& );

        int count( const elemType &value ) const;
    private:
        int       _size;
        elemType *_parray;
        elemType  _min;
        elemType  _max;
    };
```

Because `elemType` now potentially represents any built-in or user-defined class, I declare the formal parameters as `const` references rather than pass them by value.

Exercise 6.2

Reimplement the Matrix class of Exercise 5.3 as a template. In addition, extend it to support arbitrary row and column size using heap memory. Allocate the memory in the constructor and deallocate it in the destructor.

The primary work of this exercise is to add general row and column support. We introduce a constructor that takes a row and a column size as arguments. The body of the constructor allocates the necessary memory from the program's free store:

```
Matrix( int rows, int columns )
      : _rows( rows ), _cols( columns )
{
    int size = _rows * _cols;
    _matrix = new elemType[ size ];
    for ( int ix = 0; ix < size; ++ix )
          _matrix[ ix ] = elemType();
}
```

`elemType` is the template parameter. `_matrix` is now an `elemType` pointer that addresses the heap memory allocated through the `new` expression.

```
template <typename elemType>
class Matrix {
public:
    // ...
private:
    int       _rows;
    int       _cols;
```

```
    elemType *_matrix;
};
```

It is not possible to specify an explicit initial value for which to set each element of the Matrix. The potential actual types that might be specified for `elemType` are simply too diverse. The language allows us, rather, to specify a default constructor:

```
_matrix[ ix ] = elemType();
```

For an `int`, this becomes `int()` and resolves to 0. For a `float`, this becomes `float()` and resolves to 0.0f. For a string, this becomes `string()` and invokes the default string constructor, and so on.

We must add a destructor to delete the heap memory acquired during the class object's construction:

```
~Matrix(){ delete [] _matrix; }
```

We also must provide a copy constructor and a copy assignment operator now. Default memberwise initialization and copy are no longer sufficient when we allocate memory in a constructor and deallocate it in a destructor. Under the default behavior, the `_matrix` pointer member of both class objects now addresses the same heap memory. This becomes particularly nasty if one object is destroyed while the other object continues to be active within the program: Its underlying matrix has been deleted! One solution is to *deep copy* the underlying matrix so that each class object addresses a separate instance:

```
template <typename elemType>
Matrix<elemType>::Matrix( const Matrix & rhs )
{
   _rows = rhs._rows; _cols = rhs._cols;
   int mat_size = _rows * _cols;
   _matrix = new elemType[ mat_size ];
   for ( int ix = 0; ix < mat_size; ++ix )
         _matrix[ ix ] = rhs._matrix[ ix ];
}

template <typename elemType>
Matrix<elemType>& Matrix<elemType>::operator=( const Matrix &rhs )
{
   if ( this != &rhs ){
        _rows = rhs._rows; _cols = rhs._cols;
        int mat_size = _rows * _cols;
        delete [] _matrix;
        _matrix = new elemType[ mat_size ];
        for ( int ix = 0; ix < mat_size; ++ix )
              _matrix[ ix ] = rhs._matrix[ ix ];
    }

    return *this;
```

```
}
```

Here is the full class declaration and the remaining member functions:

```
#include <iostream>
template <typename elemType>
class Matrix
{
    friend Matrix<elemType>
        operator+( const Matrix<elemType>&, const Matrix<elemType>& );

    friend Matrix< elemType >
        operator*( const Matrix<elemType>&, const Matrix<elemType>& );

public:
    Matrix( int rows, int columns );
    Matrix( const Matrix& );
    ~Matrix();
    Matrix& operator=( const Matrix& );

    void operator+=( const Matrix& );
    elemType& operator()( int row, int column )
        { return _matrix[ row * cols() + column ]; }

    const elemType& operator()( int row, int column ) const
        { return _matrix[ row * cols() + column ]; }

    int rows() const { return _rows; }
    int cols() const { return _cols; }

    bool same_size( const Matrix &m ) const
         { return rows() == m.rows() && cols() == m.cols(); }

    bool comfortable( const Matrix &m ) const
         { return ( cols() == m.rows() ); }

    ostream& print( ostream& ) const;

protected:
    int  _rows;
    int  _cols;
    elemType *_matrix;
};

template <typename elemType>
inline ostream&
operator<<( ostream& os, const Matrix<elemType> &m )
    { return m.print( os ); }

// end of Matrix.h
```

```
template <typename elemType>
Matrix< elemType >
operator+( const Matrix<elemType> &m1, const Matrix<elemType> &m2 )
{
    // make sure m1 & m2 are same size
    Matrix<elemType> result( m1 );
    result += m2;
    return result;
}

template <typename elemType>
Matrix<elemType>
operator*( const Matrix<elemType> &m1, const Matrix<elemType> &m2 )
{
    // m1's columns must equal m2's rows ...
    Matrix<elemType> result( m1.rows(), m2.cols() );
    for ( int ix = 0; ix < m1.rows(); ix++ ) {
        for ( int jx = 0; jx < m1.cols(); jx++ ) {
            result( ix, jx ) = 0;
            for ( int kx = 0; kx < m1.cols(); kx++ )
                result( ix, jx ) += m1( ix, kx ) * m2( kx, jx );
        }
    }
    return result;
}

template <typename elemType>
void Matrix<elemType>::operator+=( const Matrix &m ){
    // make sure m1 & m2 are same size
    int matrix_size = cols() * rows();
    for ( int ix = 0; ix < matrix_size; ++ix )
        ( *( _matrix + ix )) += ( *( m._matrix + ix ));
}

template <typename elemType>
ostream& Matrix<elemType>::print( ostream &os ) const {
    int col = cols();
    int matrix_size = col * rows();
    for ( int ix = 0; ix < matrix_size; ++ix ){
        if ( ix % col == 0 ) os << endl;
        os << ( *( _matrix + ix )) << ' ';
    }
    os << endl;
    return os;
}
```

Here is a small program to exercise our Matrix class template:

```
int main()
```

```
{
   ofstream log( "C:\\My Documents\\log.txt" );
   if ( ! log )
      { cerr << "can't open log file!\n"; return; }

   Matrix<float> identity( 4, 4 );
   log << "identity: " << identity << endl;
   float ar[16]={ 1., 0., 0., 0., 0., 1., 0., 0.,
                  0., 0., 1., 0., 0., 0., 0., 1. };

   for ( int i = 0, k = 0; i < 4; ++i )
         for ( int j = 0; j < 4; ++j )
               identity( i, j ) = ar[ k++ ];
   log <<  "identity after set: " << identity << endl;

   Matrix<float> m( identity );
   log << "m: memberwise initialized: " << m << endl;

   Matrix<float> m2( 8, 12 );
   log << "m2: 8x12: " <<  m2  << endl;

   m2 = m;
   log << "m2 after memberwise assigned to m: "
       << m2 << endl;

   float ar2[16]={ 1.3, 0.4, 2.6, 8.2, 6.2, 1.7, 1.3, 8.3,
                   4.2, 7.4, 2.7, 1.9, 6.3, 8.1, 5.6, 6.6 };

   Matrix<float> m3( 4, 4 );
   for ( int ix = 0, kx = 0; ix < 4; ++ix )
          for ( int j = 0; j < 4; ++j )
                m3( ix, j ) = ar2[ kx++ ];

   log << "m3: assigned random values: " << m3 << endl;

   Matrix<float> m4 = m3 * identity; log << m4 << endl;
   Matrix<float> m5 = m3 + m4; log << m5 << endl;

   m3 += m4; log << m3 << endl;
}
```

When compiled and executed, the program generates the following output:

```
identity:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

identity after set:
```

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

m: memberwise initialized:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

m2: 8x12:
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
m2 after memberwise assigned to m:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

m3: assigned random values:
1.3 0.4 2.6 8.2
6.2 1.7 1.3 8.3
4.2 7.4 2.7 1.9
6.3 8.1 5.6 6.6

1.3 0.4 2.6 8.2
6.2 1.7 1.3 8.3
4.2 7.4 2.7 1.9
6.3 8.1 5.6 6.6

2.6 0.8 5.2 16.4
12.4 3.4 2.6 16.6
8.4 14.8 5.4 3.8
12.6 16.2 11.2 13.2

2.6 0.8 5.2 16.4
12.4 3.4 2.6 16.6
8.4 14.8 5.4 3.8
12.6 16.2 11.2 13.2
```

Exercise 7.1

The following function provides absolutely no checking of either possible bad data or the possible failure of an operation. Identify all the things that might possibly go wrong within the function (in this exercise, we don't yet worry about possible exceptions raised).

```
int *alloc_and_init( string file_name )
{
    ifstream infile( file_name );
    int elem_cnt;
    infile >> elem_cnt;
    int *pi = allocate_array( elem_cnt );

    int elem;
    int index = 0;
    while ( infile >> elem )
            pi[ index++ ] = elem;

    sort_array( pi, elem_cnt );
    register_data( pi );

    return pi;
}
```

The first error is a type violation: The ifstream constructor requires a `const char*` and not a string. To retrieve the C-style character string representation, we invoke the `c_str()` string member function:

```
ifstream infile( file_name.c_str() );
```

Following the definition of `infile`, we should check that it opened successfully:

```
if ( ! infile ) // open failed ...
```

If `infile` did open successfully, the third statement executes but may not succeed. For example, if the file contained text, the attempt to place an element in `elem_cnt` fails. Alternatively, it is possible for the file to be empty.

```
if ( ! infile ) // gosh, the read failed
```

Whenever we deal with pointers, we must be concerned as to whether they actually address an object. If `allocate_array()` was unable actually to allocate the array, `pi` is initialized to 0. We must test that:

```
if ( ! pi ) // geesh, allocate_array() didn't really
```

The assumption of the program is that `elem_cnt` represents a count of the elements contained within the file. `index` is unlikely to overflow the array. However, we cannot guarantee that `index` is never greater than `elem_cnt` unless we check.

Exercise 7.2

The following functions invoked in `alloc_and_init()` raise the following exception types if they should fail:

```
allocate_array() noMem
sort_array()     int
register_data()  string
```

Insert one or more try blocks and associated catch clauses where appropriate to handle these exceptions. Simply print the occurrence of the error within the catch clause.

Rather than surround each individual function invocation with a separate try block, I have chosen to surround the entire set of calls with a single try block that contains three associated catch clauses:

```
int *alloc_and_init( string file_name )
{
   ifstream infile( file_name.c_str() );
   if ( ! infile )return 0;

   int elem_cnt;
   infile >> elem_cnt;
   if ( ! infile ) return 0;

   try {
      int *pi = allocate_array( elem_cnt );
      int elem;
      int index = 0;
      while ( infile >> elem && index < elem_cnt )
               pi[ index++ ] = elem;

      sort_array( pi, elem_cnt );
      register_data( pi );
   }
   catch( const noMem &memFail ) {
         cerr << "alloc_and_init(): allocate_array failure!\n"
              << memFail.what() << endl;
         return 0;
   }
   catch( int &sortFail ) {
         cerr << "alloc_and_init(): sort_array failure!\n"
              << "thrown integer value: " << sortFail << endl;
         return 0;
   }
   catch( string &registerFail ) {
         cerr << "alloc_and_init(): register_data failure!\n"
              << "thrown string value: "
               << registerFail << endl;
         return 0;
   }
```

```
    return pi; // reach here only if no throw occurred ...
}
```

Exercise 7.3

Add a pair of exceptions to the Stack class hierarchy of Exercise 6.2 to handle the cases of attempting to pop a stack that is empty and attempting to push a stack that is full. Show the modified `pop()` and `push()` member functions.

We'll define a PopOnEmpty and a PushOnFull pair of exception classes to be thrown, respectively, in the `pop()` and `push()` Stack member functions. These classes no longer need to return a success or failure value:

```
void pop( elemType &elem )
{
     if ( empty() )
          throw PopOnEmpty();
     elem = _stack[ --_top ];
     _stack.pop_back();
}
void push( const elemType &elem ){
      if ( ! full() ){
           _stack.push_back( elem );
           ++_top;
           return;
      }
      throw PushOnFull();
}
```

To allow these two Stack class exceptions to be caught in components that do not explicitly know about PopOnEmpty and PushOnFull exception classes, the class exceptions are made part of a StackException hierarchy that is derived from the standard library logic_error class.

The logic_error class is derived from exception, which is the root abstract base class of the standard library exception class hierarchy. This hierarchy declares a virtual function `what()` that returns a `const char*` identifying the exception that has been caught.

```
class StackException : public logic_error {
public:
    StackException( const char *what ) : _what( what ){}
    const char *what() const { return _what.c_str(); }
protected:
    string _what;
};

class PopOnEmpty : public StackException {
public:
```

```
   PopOnEmpty() : StackException( "Pop on Empty Stack" ){}
};

class PushOnFull : public StackException {
public:
    PushOnFull() : StackException( "Push on Full Stack" ){}
};
```

Each of the following catch clauses handles an exception of type PushOnFull:

```
catch( const PushOnFull &pof )
     { log( pof.what() ); return; }

catch( const StackException &stke )
     { log( stke.what() ); return; }

catch( const logic_error &lge )
     { log( lge.what() ); return; }

catch( const exception &ex )
     { log( ex.what() ); return; }
```