makefile:
```
words: main.o words.o
	g++ -g -Wall main.o words.o -o words

main.o: main.cpp words.h
	g++ -c -g -Wall main.cpp

words.o: words.cpp words.h
	g++ -c -g -Wall words.cpp

clean:
	rm -rf *.o words
```

top of file comment:
```
/* -------------------------------------------------
  Filename:  naninani.h/naninani.cpp
  Author: Robert Speller
  Date: 7th January 2013
  ------------------------------------------------- */
```

header file:
```
#ifndef WORDS_H
#define WORDS_H
...
const int SOUNDEX_SIZE = 4;
//4 characters in soundex
...
#endif  // WORDS_H
```

```
while( *sentence != '\0' )
//could use this instead:
while( *sentence )
```

assert() cassert

modulus operator %

static_cast<double>(operand)

(double) a / b

e.g. number = 5 + rand() % 11; // produces a random number between 5 and 15 inclusive.

enum Status (CONTINUE, WON, LOST);

Local variables declared **static** maintain their values when the function in which they are declared is exited.

in_stream.close(); out_stream.close();
should be included when you are finished taking input from/writing output to variables in_stream and out_stream.

floor(x + 0.5)
this always round to the nearest integer.

```
struct Node {
int number;
char name[ 20 ];
Node *ptr_to_next_node;
};
```

```
int *ptr_a;                    int *ptr_b;
ptr_a = new int;               ptr_b = new int [6];
*ptr_a = 6;                    ptr_b[3] = 8;
delete ptr_a;                  delete [] ptr_b;
```

Library Type	Description	Header
istream	Standard input (cin)	iostream
ostream	Output (cout, cerr, clog)	iostream
ifstream	Input file	fstream
ofstream	Output file	fstream
string	Sequence of char	string
vector<T>	Expandable array/stack of T	vector
deque<T>	Array/double ended queue	deque
list<T>	List/stack/queue of T	list
map<T1,T2>	Associative mapping of T1 to T2	map
set<T1>	A map with keys only	set
pair<T1,T2>	Two objects of type T1 and T2	map or utility
iterator	Pointer into a container	(Included with container)

Built-in	Description
int x;	Fastest integer type (16-32 bits), also short, long, unsigned
char x;	8-bit character, '\0' to '\xFF' or -128 to 127
double x;	64 bit real + or - 1.8e308, 14 significant digits, also float
bool x;	true or false

Modifiers	Description
const T x;	Non-modifiable object
T& y=x;	Reference, y is an alias for x, which both have type T
T f(...) {...}	Defines f as a function returning T
T* p;	Pointer to T (*p is a T object)
T a[N];	Array of N elements of T, a[0] to a[N-1]
static T x;	Place x in data segment

<cstring>
```
strcpy(dst, src);
// Copy src to dst. Not bounds checked
strcat(dst, src);
// Concatenate to dst. Not bounds checked
strcmp(s1, s2);
// Compare, <0 if s1<s2, 0 if s1==s2, >0 if s1>s2
strncpy(dst, src, n);
// Copy up to n chars, also strncat(), strncmp()
strlen(s);            // Length of s not counting \0
strchr(s,c); strrchr(s,c);
// Address of first/last char c in s or 0
strstr(s, sub);
// Address of first substring in s or 0
```

<cmath>
```
Functions take double and return double.
pow(x, y); sqrt(x);     // x to the y, square root
ceil(x); floor(x);
// Round up or down (as a double)
fabs(x); fmod(x, y);    // Absolute value, x mod y
```

<cctype>
```
Character tests take a char c and return bool.
isalnum(c);           // Is c a letter or digit?
isalpha(c); isdigit(c);  // Is c a letter?  Digit?
islower(c); isupper(c);
// Is c lower case?  Upper case?
isspace(c);           // Is whitespace?
ispunct(c);
// Is printing except space, letter, or digit?
isxdigit(c);          // Is hexadecimal digit?
c=tolower(c); c=toupper(c);
// Convert c to lower/upper case
```

<cstdlib>
```
Miscellaneous functions. s is type char*, n is int
atoi(s); atol(s); atof(s);
// Convert char* s to int, long, double e.g. atof("3.5")
abs(x); labs(x);
// Absolute value of numeric x as int, long
rand();
// Pseudo-random int from 0 to RAND_MAX (at least 32767)
srand(n);
// Initialize rand(), e.g. srand(time(0)); time() belongs to <ctime>
exit(n);
// Kill program, return status n, e.g. exit(0);
```

ASCIIs:
```
8-13:  \b\t\n\v\f\r  (bell, tab, newline, vertical tab, formfeed, return)
32-47:  !\"#$%&\'()*+,-./          (32=space, \' and \" are one char)
48-63:  0123456789:;<=>\?          (\? is one char)
64-95:  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_  (\\ is one char)
96-126: `abcdefghijklmnopqrstuvwxyz{|}~
```

<iostream>
```
In the following, in is an istream (cin), out is an ostream (cout, cerr, clog), i is int, c is char, and cp is char*.
in >> x;
// Read 1 word to numeric, string, or char* x, return in
in.get();
// Read 1 char (0-255) or EOF (-1) as an int
in.get(c);          // Read 1 char into c, return in
in.putback();  // Put back last char read, return in
in.getline(cp, i);
// Read up to i chars into char cp[i] or until '\n', return in
in.getline(cp, i, c);
// Read to c instead of '\n', return in
getline(in, s);
// Read up to '\n' into string s, return in
in.good();          // true if no error or EOF
bool(in);           // in.good();
in.bad();           // true if unexpected char in formatted input
in.clear();         // Allow more input after bad, or throw an ios::failure
in.eof();           // true if end of file (has been read in by get already!)
in.fail();          // true if system input error

out << x;
// Formatted output, redirected with >
out << endl;          // Print '\n' and flush
```

<iomanip>
```
Defines manipulators for formatted output of numeric types. They have no effect on strings.
setw() applies only to the next object printed, but the others remain in effect until changed.
out << setw(i);
// Pad next output to i chars, then back to 0
out << setfill(c);     // Pad with c (default ' ')
out << setprecision(i);
// Use i significant digits for all float, double
cout << setw(6) << setprecision(3) << setfill('0') << 3.1;
// print "003.10"
```

```
ifstream in_stream;
in_stream.open("Lecture_4");
if (in_stream.fail())
{
   cout << "Sorry, the file couldn't be opened!\n";
exit(1); }...
```

<string>
```
A string is like an array of char, but it also supports copying, assignment, and comparison, and its size may be set or changed at run time. '\0' has no special meaning. There is implicit conversion from char* to string in mixed type expressions.
string()        // Empty string
string(cp)      // Convert char* cp to string
string(n, c)    // string of n copies of char c
s=s2            // Assign char* or string s2 to string s
s1<s2
// Also ==, !=, >, <=, >=, either s1 or s2 may be char*
s.size()        // Length of string s
string::size_type
// Type of s.size(), usually unsigned int
s.empty()       // True if s.size() == 0
s[i]
// i'th char, 0 <= i < s.size() (unchecked), may be assigned to
s.at(i)
// s[i] with bounds check, throws out_of_range
s1+s2
// Concatenate strings, either s1 or s2 may be char or char*
s+=s2
// Append string, char, or char* s2 to string s
s.c_str()
// string s as a const char* with trailing '\0'
s.substr(i, j)
// Substring of string s of length j starting at s[i]
s.substr(i)     // Substring from s[i] to the end
s.find(s2)
// Index of char, char*, or string s2 in s, or string::npos if not found
s.rfind(s2)     // Index of last occurrence of s2 in s
s.find_first_of(s2)
// Index of first char in s that occurs in s2
s.find_last_of(s2)
// Index of last char in s that occurs in s2
s.find_first_not_of(s2)
// Index of first char in s not found in s2
s.find_last_not_of(s2)
// Index of last char in s not found in s2
s.replace(i, j, s2)   // Replace s.substr(i, j) with s2
s.size() should be converted to int to avoid unsigned comparison.
string s(3,'a');  // "aaa"
s += "b"+s;       // "aaabaaa"
for (int i=0; i!=int(s.size()); ++i) {
// print s one char at a time
  cout << s[i];
s.size() > -1;    // false!  -1 is converted to unsigned
```

```
if ( ! isalpha(*surname) ) {
  cout << "Error: Invalid surname entered!\n";
  exit(1);
}
```

```
bool compare( const char *str1, const char *str2 ) {
 if ( *str1 == '\0' && *str2 == '\0' )
  return true;
```

```
  else if ( *str1 != *str2 )
    return false;
  else
    return compare(++str1,++str2);
}
```

```
char word[512];
...
void get_word( const char * sentence, char* word )
{       int n = 0;
        while ( isalpha(*sentence) )
                word[n++] = *sentence++;
        word[n] = '\0';
}
```

```
bool is_prefix( const char *str1, const char *str2) {
  if ( *str1 == '\0' )
    return true;
  else if ( *str1 != *str2 )
    return false;
  else
    return is_prefix( ++str1, ++str2 );
}
```

```
int substring_position( const char *str1, const char
*str2 ) {
  int n = 0;
  do {
    if ( is_prefix( str1, str2 ) )
      return n;
    ++n; }
  while ( *str2++ != '\0' );
  return -1;
}
```

```
/**
 *  Checks if a string is contained in another string.
 *  @a: substring to match
 *  @b: string to be checked
 *
 *  Returns a zero-indexed position to the first
occurrence in @b of the
 *  string @a; otherwise returns -1.
 */
int substring_position2( const char *a, const char
*b ) {         return(a=strstr(b,a))?a-b:-1; }
```

```
void reverse( const char *str1, char *str2 ) {
  const char *str1_end = str1;  //a second pointer
pointing to str1

  while (*str1_end != '\0' )    //point to the sentinel
character
    ++str1_end;
  --str1_end;                   //point to the previous
character

  while ( str1_end >= str1 )    //copy characters to
str2 going backwards
    *str2++ = *str1_end--;      //through str1 until
reaching the start.
  *str2 = '\0';                 //add the sentinel character
to str2
}
```

```
bool compare( const char *one, const char *two ) {
  while ( !( isalpha(*one) || *one == '\0') ) //skip to
next letter or sentinel
    ++one;
  while ( !( isalpha(*two) || *two == '\0') ) //skip to
next letter or sentinel
    ++two;

  if ( *one == '\0' && *two == '\0' )        //both
reached sentinel
    return true;

  else if ( *one == '\0' || *two == '\0' ||   //only one
reached sentinel
            toupper(*one) != toupper(*two) )  //or
characters not equivalent
    return false;

  else                                  //equivalent characters
    return compare(++one, ++two);
}
```

```
bool palindrome( const char *sentence ) {
  char reversed[512];
  reverse( sentence, reversed );
  return compare(sentence,reversed);
}
```

```
void get_letters( const char *string, int
*letters_array ) {
  //increment the array element for the particular
letter
  if (isalpha(*string))
    ++letters_array[ toupper(*string) - 'A' ];

  // recurse through string character by character
  if ( *string != '\0' )
    get_letters( ++string, letters_array );
}
```

```
bool equal_int_arrays( const int *array1, const int *array2,
const int size ) {
  for ( int n = 0; n < size; n++ )
    if ( array1[n] != array2[n] )
      return false;
  return true;
}
```

```
bool anagram( const char *str1, const char *str2 ) {
  //integer arrays to hold the totals of each letter for both
strings
  int str1_letters[LETTERS_IN_ALPHABET] = {0}; //initialise
to 0's
  int str2_letters[LETTERS_IN_ALPHABET] = {0};
  get_letters( str1, str1_letters );
  get_letters( str2, str2_letters );
  return equal_int_arrays(str1_letters, str2_letters,
LETTERS_IN_ALPHABET);
}
```

```
bool get_word(const char *input_line, int word_number,
char *output_word) {
  char *output_start = output_word;
  int words = 0;

  if (word_number < 1) {
    *output_word = '\0';
    return false;
  }

  do {
    while (*input_line && !isalnum(*input_line))
      input_line++;

    if (*input_line == '\0')
      break;

    output_word = output_start;
    while (*input_line && (isalnum(*input_line) ||
*input_line=='\'')) {
      *output_word = toupper(*input_line);
      output_word++;
      input_line++;
    }
    *output_word = '\0';

    if (++words == word_number)
      return true;

  } while (*input_line);

  *output_start = '\0';
  return false;
}
```

```
int count_words(const char *line) {
  char word[512];
  int n;
  for (n=0; get_word(line, n+1, word); n++);
  return n;
}
```

```
bool includes_vowel( const char *word ) {
  const char * vowels = "AEIOU";
  for (unsigned int n=0; n<strlen(vowels),n++) {
    if (strchr(word, vowels[n]))
      return true;
  }
  return false;
}
```

```
const int MAX_WORD_LENGTH = 80;

/* definition of a node */
struct Node;
typedef Node *Node_ptr;

struct Node
{
        char word[MAX_WORD_LENGTH];
        Node_ptr ptr_to_next_node;
};

int main()
{
        Node_ptr my_list = NULL;

        assign_list(my_list);

        cout << "\nTHE LIST IS NOW:\n";
        print_list(my_list);

        return 0;
}
```

```
/* END OF MAIN PROGRAM */

/* Function to assign a linked list to "a_node" */
void assign_list(Node_ptr &a_list)
{
        Node_ptr current_node, last_node;

        assign_new_node(a_list);
        cout << "Enter first word (or '.' to end list): ";
        cin >> a_list->word;
        if (!strcmp(".",a_list->word))
        {
                delete a_list;
                a_list = NULL;
        }
        current_node = a_list;

        while (current_node != NULL)
        {
                assign_new_node(last_node);
                cout << "Enter next word (or '.' to end
list): ";
                cin >> last_node->word;
                if (!strcmp(".",last_node->word))
                {
                        delete last_node;
                        last_node = NULL;
                }
                current_node->ptr_to_next_node =
last_node;
                current_node = last_node;
        }
}
/* END OF FUNCTION DEFINITION */

/* Function to assign a new dynamic node variable to "a_node" */
void assign_new_node(Node_ptr &a_node)
{
        a_node = new Node;
        if (a_node == NULL)
        {
                cout << "sorry - no more memory\n";
                exit(1);
        }
}
/* END OF FUNCTION DEFINITION */

/* Function to print the strings in list "a_node" */
void print_list(Node_ptr a_node)
{
        while (a_node != NULL)
        {
                cout << a_node->word << " ";
                a_node = a_node->ptr_to_next_node;
        }
}
/* END OF FUNCTION DEFINITION */
```

```
void selection_sort(int a[], int length)
    {   for (int count = 0 ; count < length - 1 ;
            count++)
        swap(a[count],
            a[minimum_from(a,count,length)]);
    }

int minimum_from(int a[], int position, int length)
    {
        int min_index = position;

        for (int count = position + 1 ; count <          length ; count +
+)
            if (a[count] < a[min_index])
                min_index = count;

        return min_index;
    }

void swap(int& first, int& second)
    {
        int temp = first;
        first = second;
        second = temp;
    }
```

```
int lookup( const char *word, const char *dictionary[] ) {
  int encoding = 0;
  while (strcmp(*dictionary,"")) {
    if (strcmp(*dictionary,word) == 0)
      return encoding;
    ++dictionary;
    ++encoding;
  }
  return -1;
}
void intToStr(const int code, char *strcode) {
  int digit1, digit2;
  digit1 = code / 10;
  digit2 = code % 10;

  strcode[0] = '0' + digit1;
  strcode[1] = '0' + digit2;
  strcode[2] = '\0';
}
void encode( const char *word, char *compressed, const char
*dictionary[] ) {
  if (isdigit(*word) || *word == '!') {
    strcpy(compressed,"!");
    strcat(compressed,word);
    return;
...
  }
}
```