1. Introduction.

ANAGRAM

Anagram Finder

by John Walker


```
#define PRODUCT "anagram"
#define VERSION "1.2"
#define REVDATE "2003-10-05"
```

2. Command line.

`ANAGRAM` is invoked with a command line as follows:

`anagram` *options* [*'target phrase'*] [*seed...*]

where *options* specify processing modes as defined below and are either long names beginning with two hyphens or single letter abbreviations introduced by a single hyphen. The *target phrase* is the phrase for which anagrams will be sought. If one or more *seed* words is given, only anagrams containing all of those words will be shown. The target and seed may be specified by options or CGI program environment variables as well as on the command line.

3. Options.

Options are specified on the command line prior to the input and output file names (if any). Options may appear in any order. Long options beginning with "`--`" may be abbreviated to any unambiguous prefix; single-letter options introduced by a single "`-`" may be aggregated.

`--all`
: Generate all anagrams in `--cgi --step` 2.

`--bail`
: Bail out after the first anagram found containing a word. In many cases this drastically reduces the time required to run the program. You can review the list of words appearing in anagrams and then request a complete list of anagrams containing "interesting" ones.

`--bindict`, `-b` *file*
: Load binary dictionary from *file*. The default binary dictionary is `wordlist.bin`.

`--cgi`
: When executed as a CGI program on a Web server, set options from form fields. These settings may be overridden by command line options which follow `--cgi`.

`--copyright`
: Print copyright information.

`--dictionary`, `-d` *file*
: Load word list from *file*. This is (only) used with the `--export` option to compile a word list into a binary dictionary. The default word list is `crossword.txt`.

`--export` *file*
: Create a binary dictionary *file* from the word list specified by the `--dictionary` option of the default `crossword.txt`.

`--help`, `-u`
: Print how-to-call information including a list of options.

`--html`
: Generate HTML output when run as a CGI program on a Web server. The HTML is based on the template file specified by the `--template` option, which defaults to a file named `template.html` in the current directory.

`--permute`, `-p` '*phrase*'
: Print all permutations of words in the given *phrase*. The phrase must be quoted so that blanks separating words are considered part of the single *phrase* argument.

`--seed`, `-s` *word*
: Find only anagrams which contain the specified *word*. You may specify as many seed words as you wish (each with a separate `--seed` option) to restrict the anagrams to those containing all of the seed words. To obtain a list of words which appear in anagrams of a phrase, specify the `--bail` option. You can also specify seed words on the command line after the options and target phrase, if any.

`--step` *number*
: Perform step *number* when operating as a CGI program on a Web server. Step 0, the default, selects non-CGI operation; the program acts as a command line utility. In step 1, a list of words appearing in anagrams is generated. Step 2 generates anagrams in which a selected word appears (or, all anagrams if the HTML template permits this option). Step 3 generates all permutations of words in a selected anagram.

`--target`, `-t` '*phrase*'
: Generate anagrams or permutations for the specified *phrase*, which must be quoted if it contains more than a single word. If no `--target` is specified, the program uses the first command line argument as the target.

`--verbose`, `-v`
: Print diagnostic information as the program performs various operations.

`--version`
: Print program version information.

4. Using `anagram` as a Web Server CGI Program.

It is possible to install `anagram` as a CGI (Common Gateway Interface) program on a Web server and thereby provide a Web-based anagram finder. **In most cases this would be extremely unwise!** It is extremely easy for an innocent user or pimple-faced denial of service moron to enter a sequence of letters for which tens of millions of anagrams exist and thereby lock your server in a CPU-intensive loop for many minutes, then clog your ISP connection with the enormous results. Having done so, the latter variety of bottom feeder will immediately write a script to submit several hundred such requests per second to your server.

You may think my view of my fellow denizens of Cyberia somewhat jaundiced, but it is based on the brutal experience of operating a public Web server since 1994, before which I was entirely too sanguine about the sanguinary intentions of some users of freely provided resources. The only circumstances in which it may make sense to install `anagram` on a Web server are on in-house Intranets where you wish to make the facility available to users without the need to port the program to all of the platforms employed by your users, and access logging and appropriate chastisement with an aluminium baseball bat is adequate to deal with abuse of the resource.

There are many different Web servers, and even commonly used servers such as Apache are installed in a multitude of different ways. Consquently, I can't provide a cookbook procedure for installing this program as a CGI resource, only templates which can serve as a point of departure. I cannot provide any support or assistance in setting up a Web-based application based on this program—if you lack the experience required, consult a Webmaster familiar with your server configuration.

First, you need to build a version of `anagram` configured to execute on your server hardware. Assuming the server has the proper compiler installed, this is just a matter of following the regular build procedure. Be sure to test the resulting program in normal command line mode to make sure it works. If your server is a minimalist "stripped" configuration without development tools (as are many "thin servers" in "server farms"), you'll have to build the program on a compatible development machine then copy it to the server(s). Be sure to test the program *on the server*—sometimes you'll discover it requires shared libraries present on the build machine but not the server; if this occurs, you'll need to either install the libraries on the server or re-build the application with statically linked libraries (which will result in a larger program which takes longer to load).

Once you have a version of the program which runs on your server, copy it to your Web server's `cgi-bin` directory. If you don't know what this means, you shouldn't be reading this section.

Now you must install a shell script which invokes the `anagram` program with the appropriate options when a client submits a request. This must provide the program the complete path for the binary dictionary and the HTML result template files. The file `cgiweb/AnagramFinder` is an example of such a script; in most cases you'll need only to change the `HTTPD` declaration at the top to adapt it to your server.

The shell script provides the location of the binary dictionary file (`--bindict`) and HTML template (`--template`). These are usually kept in a subdirectory of your `cgi-bin` directory, but may be installed in any location accessible by CGI programs. Copy the `wordlist.bin` binary dictionary from the main distribution directory and the `template.html` file from the `cgiweb` subdirectory to the designated locations. You'll want to review the latter file to adapt top links, etc. for your own site.

Finally, adapt the `cgiweb/index.html` page for your site and install it as the page where users start the anagram generation process. Test, fix the myriad obscure bugs attendant to bringing up any new CGI resource, and you're in business.

5. Program global context.

⟨ Preprocessor definitions ⟩
⟨ System include files 52 ⟩
⟨ Program implementation 6 ⟩

6. The following classes are defined and their implementations provided.

⟨ Program implementation 6 ⟩ ≡
⟨ Global variables 35 ⟩
⟨ Class definitions 7 ⟩
⟨ Command line arguments 53 ⟩
⟨ Class implementations 8 ⟩
⟨ Global functions 34 ⟩
⟨ Main program 25 ⟩

This code is used in section 5.

7. Dictionary Word.

A *dictionaryWord* is a representation of a string of letters which facilitates operations common in word games. When a *dictionaryWord* is initialised, auxiliary storage is set to a count of letters (neglecting case and diacritical marks), and of the type of characters comprising the string. Operators and methods permit quick relational tests between objects of this type.

```
⟨ Class definitions 7 ⟩ ≡
  class dictionaryWord {
  public:
    string text;      /* The word itself */
    unsigned char letterCount[26];      /* Flattened letter count */
        /* The following fields contain counts of characters of various classes in the word. You can use
        them by themselves to test whether a character of the given type is present in the word. */
    unsigned int upper,      /* Upper case ASCII letters */
    lower,      /* Lower case ASCII letters */
    digits,      /* Numeric digits */
    spaces,      /* White space */
    punctuation,      /* Other ASCII characters */
    ISOupper,      /* Upper case ISO letters */
    ISOlower,      /* Lower case ISO letters */
    ISOpunctuation;      /* Other ISO characters */
    dictionaryWord(string s = "")
    {
      set(s);
    }
    dictionaryWord(int i)
    {
      text = "";
      memset(letterCount, 0, sizeof letterCount);
      upper = lower = digits = spaces = punctuation = ISOupper = ISOlower = ISOpunctuation = 0;
    }
    void set(string s = "")
    {
      text = s;
      update( );
    }
    string get(void)
    {
      return text;
    }
    unsigned int length(void) const
    {      /* Return length of word */
      return text.length( );
    }
    void noBlanks(void)
    {      /* Delete blanks */
      string::iterator ep = remove_if (text.begin( ), text.end( ), &dictionaryWord::is_iso_space);
      text.resize(ep − text.begin( ));
      update( );
    }
    void onlyLetters(void)
```

```
{   /* Delete all non-letters */
  string::iterator ep = remove_if(text.begin(), text.end(), &dictionaryWord::is_non_iso_letter);
  text.resize(ep - text.begin());
  update();
}
void toLower(void)
{   /* Convert to lower case */
  transform(text.begin(), text.end(), text.begin(), &dictionaryWord::to_iso_lower);
  update();
}
void toUpper(void)
{   /* Convert to upper case */
  transform(text.begin(), text.end(), text.begin(), &dictionaryWord::to_iso_upper);
  update();
}
void ISOtoASCII(void);
void describe(ostream &os = cout);
bool operator≤(dictionaryWord &w);
bool operator > (dictionaryWord &w);
bool operator≥(dictionaryWord &w);
bool operator < (dictionaryWord &w);
bool operator≡(dictionaryWord &w);
bool operator≠(dictionaryWord &w);
bool contained(const dictionaryWord *wbase, const dictionaryWord *candidate);
bool contained(const dictionaryWord *wbase, unsigned char *candidate);
dictionaryWord operator+(dictionaryWord &w);
dictionaryWord operator−(dictionaryWord &w);
dictionaryWord operator+=(dictionaryWord &w);
void exportToBinaryFile(ostream &os);
protected:
void countLetters(void);
void update(void)
{
  memset(letterCount, 0, sizeof letterCount);
  upper = lower = digits = spaces = punctuation = ISOupper = ISOlower = ISOpunctuation = 0;
  countLetters();
}
⟨ Transformation functions for algorithms 16 ⟩;
};
```

See also sections 18, 19, and 21.

This code is used in section 6.

8. The *countLetters* method prepares the letter count table, counting characters by class as it goes.

```
⟨ Class implementations 8 ⟩ ≡
  void dictionaryWord::countLetters(void)
  {
    const char *cp = text.c_str();
    unsigned int c;
    G_clear();
    while ((c = *cp++) ≠ 0) {
      if (c ≥ 'A' ∧ c ≤ 'Z') {
        letterCount[c − 'A']++;
      }
      else if (c ≥ 'a' ∧ c ≤ 'z') {
        letterCount[c − 'a']++;
      }
      (*(letter_category[c]))++;
#ifdef ISO_NEEDED
      if (c ≥ #A0) {
        const char *flat = flattenISO[((unsigned char) c) − #A0];
        while ((c = *flat++) ≠ 0) {
          if (islower(c)) {
            c = toupper(c);
          }
          letterCount[c − 'A']++;
        }
      }
#endif
    }
    lower = G_lower;
    upper = G_upper;
    digits = G_digits;
    spaces = G_spaces;
    punctuation = G_punctuation;
    ISOlower = G_ISOlower;
    ISOupper = G_ISOupper;
    ISOpunctuation = G_ISOpunctuation;
  }
```

See also sections 9, 10, 11, 12, 13, 14, 15, 20, 23, and 24.

This code is used in section 6.

9. Sometimes we wish to explicitly flatten ISO accented characters in a string to their ASCII equivalents. This method accomplishes this. Note that this makes sense primarily for accented letters; other characters are transformed more or less plausibly, but the results won't make sense for most word games.

```
⟨ Class implementations 8 ⟩ +≡
  void dictionaryWord::ISOtoASCII(void)
  {
    for (string::iterator p = text.begin(); p ≠ text.end(); p++) {
      if (((unsigned char) *p) ≥ #A0) {
        int n = p − text.begin();
        unsigned int c = ((unsigned char) *p) − #A0;
        text.replace(p, p + 1, flattenISO[c]);
        p = text.begin() + n + (strlen(flattenISO[c]) − 1);
      }
    }
  }
```

10. The *describe* method writes a human-readable description of the various fields in the object to the designated output stream, which defaults to *cout*.

```
⟨ Class implementations 8 ⟩ +≡
  void dictionaryWord::describe(ostream &os)
  {
    os ≪ text ≪ endl;
    os ≪ "␣␣Total␣length:␣" ≪ length() ≪ "␣characters." ≪ endl;
    for (unsigned int i = 0; i < (sizeof letterCount); i++) {
      if (letterCount[i] > 0) {
        cout ≪ "␣␣" ≪ static_cast⟨char⟩(i + 'a') ≪ "␣␣" ≪ setw(2) ≪
            static_cast⟨int⟩(letterCount[i]) ≪ endl;
      }
    }
    os ≪ "␣␣ASCII:␣Letters:␣" ≪ (upper + lower) ≪ "␣␣(Upper:␣" ≪ upper ≪ "␣␣Lower:␣" ≪
        lower ≪ ").␣␣Digits:␣" ≪ digits ≪ "␣␣Punctuation:␣" ≪ punctuation ≪ "␣␣Blanks:␣" ≪
        spaces ≪ endl;
    os ≪ "␣␣ISO:␣Letters:␣" ≪ (ISOupper + ISOlower) ≪ "␣␣(Upper:␣" ≪ ISOupper ≪
        "␣␣Lower:␣" ≪ ISOlower ≪ ").␣␣Punctuation:␣" ≪ ISOpunctuation ≪ endl;
  }
```

11. This method writes a binary representation of the word to an output stream. This is used to create the binary word database used to avoid rebuilding the letter and character category counts every time. Each entry begins with the number of characters in the word follows by its text. After this, 26 single byte letter counts are written, followed by 8 bytes containing the character category counts.

```
⟨ Class implementations 8 ⟩ +≡
  void dictionaryWord::exportToBinaryFile(ostream &os){ unsigned char c;
#define outCount(x)c = (x); os.put(c)
      outCount(text.length( ));
      os.write(text.data( ), text.length( ));
      for (unsigned int i = 0; i < sizeof letterCount; i++) {
        os.put(letterCount[i]);
      }
      outCount(upper);
      outCount(lower);
      outCount(digits);
      outCount(spaces);
      outCount(punctuation);
      outCount(ISOupper);
      outCount(ISOlower);
      outCount(ISOpunctuation);
#undef outCount
      }
```

12. We define the relational operators in a rather curious way. They test whether a word is "contained" within another in terms of the count of letters it contains. A word is *less* than another if it contains fewer of every letter. For example "*bet*" is less than "*beet*" because it contains only one "*e*". This definition of magnitude is extremely useful in a variety of word games, as it permits quick rejection of inapplicable solutions.

Since the various operator implementations differ only in the relational operator applied across the *letterCount* arrays of the two arguments, we can stamp out the code cookie-cutter style with a macro.

```
#define dictionaryWordComparisonOperator(op)
          bool dictionaryWord::operator op (dictionaryWord &w)
      {
        for (unsigned int i = 0; i < sizeof letterCount; i++) {
          if (¬(letterCount[i] op w.letterCount[i])) {
            return false;
          }
        }
        return true;
      }
⟨ Class implementations 8 ⟩ +≡
  dictionaryWordComparisonOperator ( < ) ;
  dictionaryWordComparisonOperator ( > ) ;
  dictionaryWordComparisonOperator ( ≡ ) ;
  dictionaryWordComparisonOperator ( ≠ ) ;
  dictionaryWordComparisonOperator ( ≤ ) ;
  dictionaryWordComparisonOperator ( ≥ ) ;
```

13. Addition and subtraction are defined as concatenation of strings for addition and deletion of characters in the right hand operand from the left hand string (with, in both cases, recomputation of the parameters). The += operator is implemented in a particularly efficient manner since we don't need to re-count the letters and categories—they may simply be summed in place.

⟨ Class implementations 8 ⟩ +≡

```
dictionaryWord dictionaryWord::operator+(dictionaryWord &w)
{
  dictionaryWord result(text + w.text);
  return result;
}
dictionaryWord dictionaryWord::operator+=(dictionaryWord &w)
{
  text += w.text;
  for (unsigned int i = 0; i < sizeof letterCount; i++) {
    letterCount[i] += w.letterCount[i];
  }
  lower += w.lower;
  upper += w.upper;
  digits += w.digits;
  spaces += w.spaces;
  punctuation += w.punctuation;
  ISOlower += w.ISOlower;
  ISOupper += w.ISOupper;
  ISOpunctuation += w.ISOpunctuation;
  return *this;
}
dictionaryWord dictionaryWord::operator-(dictionaryWord &w)
{
  dictionaryWord result = *this;
  for (string::iterator p = w.text.begin(); p ≠ w.text.end(); p++) {
    string::size_type n = result.text.find(*p);
    if (n ≠ string::npos) {
      result.text.erase(n, 1);
    }
  }
  return result;
}
```

14. This is a kludge. When searching for anagrams and other such transformations, we often wish to know whether a candidate word or phrase is "contained" within another. Containment is defined as having counts for all letters less than or equal to that of the target phrase. This test can be accomplished straightforwardly by generating a **dictionaryWord** for the candidate and then testing with our relational operators, but that clean approach can consume enormous amounts of time when you're testing lots of candidates. This special purpose method is passed pointers to two **dictionaryWord**s which are logically concatenated (which is what we need when testing potential anagrams—if you're testing only one letter sequence, pass a pointer to an empty word for the second argument. The argument words are tested for containment against the word in the object and a Boolean result is immediately returned. No recomputation of letter counts need be done.

```
⟨ Class implementations 8 ⟩ +≡
  bool dictionaryWord::contained(const dictionaryWord *wbase, const dictionaryWord
          *candidate)
  {
    for (unsigned int i = 0; i < sizeof letterCount; i++) {
      if ((wbase→letterCount[i] + candidate→letterCount[i]) > letterCount[i]) {
        return false;
      }
    }
    return true;
  }
```

15. This is a version of the *contained* method which directly uses a pointer into the binary dictionary. This avoids the need to construct a **dictionaryWord** for the candidate, but rather tests the letter counts "just sitting there" in the binary dictionary item.

```
⟨ Class implementations 8 ⟩ +≡
  bool dictionaryWord::contained(const dictionaryWord *wbase, unsigned char *candidate)
  {
    unsigned char *lc = binaryDictionary::letterCount(candidate);
    for (unsigned int i = 0; i < sizeof letterCount; i++) {
      if ((wbase→letterCount[i] + lc[i]) > letterCount[i]) {
        return false;
      }
    }
    return true;
  }
```

16. The following are simple-minded transformation functions passed as arguments to STL algorithms for various manipulations of the text.

⟨ Transformation functions for algorithms 16 ⟩ ≡

```
static bool is_iso_space(char c)
{
  return isspace(c) ∨ (c ≡ '\xA0');
}
static bool is_non_iso_letter(char c)
{
  return ¬isISOalpha(c);
}
static char to_iso_lower(char c)
{
  return toISOlower(c);
}
static char to_iso_upper(char c)
{
  return toISOupper(c);
}
```

This code is used in section 7.

17. Dictionary.
A *dictionary* is simply a collection of **dictionaryWord**s. You can extract individual words or perform wholesale operations on dictionaries. The *dictionary* is defined as an extension of the template class **vector**⟨**dictionaryWord**⟩, inheriting and making public all of its functionality. All of the STL methods and algorithms which apply to a **vector** will work with a *dictionary* in the same manner.

18. First of all, we need to introduce a silly little class which compares two dictionary entries by length which is needed by the *sortByDescendingLength* method defined below.

```
⟨ Class definitions 7 ⟩ +≡
  class dlencomp {
  public:
    int operator( )(const dictionaryWord &a, const dictionaryWord &b) const
    {
      return a.length( ) > b.length( );
    }
  };
```

19. Okay, now that *that's* out of the way, we can get on with the serious business. Here's the *dictionary* class definition.

```
⟨ Class definitions 7 ⟩ +≡
  class dictionary : public vector⟨dictionaryWord⟩ {
  public:
    void loadFromFile(istream &is, bool punctuationOK = true, bool digitsOK = true)
    {
      string s;
      while (getline(is, s)) {
        dictionaryWord w(s);
        if ((punctuationOK ∨ ((w.punctuation ≡ 0) ∧ (w.spaces ≡ 0) ∧ (w.ISOpunctuation ≡
            0))) ∧ (digitsOK ∨ (w.digits ≡ 0))) {
          push_back(dictionaryWord(s));
        }
      }
      if (verbose) {
        cerr ≪ "Loaded␣" ≪ size( ) ≪ "␣words␣from␣word␣list." ≪ endl;
      }
    }
    void describe(ostream &os = cout)
    {
      vector⟨dictionaryWord⟩::iterator p;
      for (p = begin( ); p ≠ end( ); p++) {
        cout ≪ p→text ≪ endl;
      }
    }
    void sortByDescendingLength(void)
    {
      stable_sort(begin( ), end( ), dlencomp( ));
    }
    void exportToBinaryFile(ostream &os);
  };
```

20. To avoid the need to reconstruct the dictionary from its text-based definition, we can export the dictionary as a binary file. This method simply iterates over the entries in the dictionary, asking each to write itself to the designated binary dictionary file. A zero byte is written at the end to mark the end of the table (this is the field which would give the length of the next word). The first four bytes of the binary dictionary file contain the number of entries in big-endian order.

```
⟨ Class implementations 8 ⟩ +≡
  void dictionary::exportToBinaryFile(ostream &os)
  {
    unsigned long nwords = size( );
    os.put(nwords ≫ 24);
    os.put((nwords ≫ 16) & #FF);
    os.put((nwords ≫ 8) & #FF);
    os.put(nwords & #FF);
    vector⟨dictionaryWord⟩::iterator p;
    for (p = begin( ); p ≠ end( ); p++) {
      p→exportToBinaryFile(os);
    }
    os.put(0);
    if (verbose) {
      cerr ≪ "Exported␣" ≪ nwords ≪ "␣to␣" ≪ os.tellp( ) ≪ "␣byte␣binary␣dictionary." ≪ endl;
    }
  }
```

21. Binary Dictionary.

The binary dictionary accesses a pre-sorted and -compiled dictionary which is accessed as a shared memory mapped file. The *raison d'être* of the binary dictionary is speed, so the methods used to access it are relatively low-level in the interest of efficiency. For example, copying of data is avoided wherever possible, either providing pointers the user can access directly or setting fields in objects passed by the caller by pointer.

The binary dictionary is created by the `--export` option of this program, by loading a word list and then writing it as a binary dictionary with the **dictionary** :: *exportToBinaryFile* method.

```
⟨ Class definitions 7 ⟩ +≡
  class binaryDictionary {
  public:
    long flen;      /* File length in bytes */
    unsigned char *dict;      /* Memory mapped dictionary */
    int fileHandle;      /* File handle for dictionary */
    unsigned long nwords;      /* Number of words in dictionary */
    static const unsigned int letterCountSize = 26, categoryCountSize = 8;
    void loadFromFile(string s)
    {
      ⟨ Bring binary dictionary into memory 22 ⟩;
    }
    binaryDictionary( )
    {
      fileHandle = −1;
      dict = Λ;
    }
    ~binaryDictionary( )
    {
#ifdef HAVE_MMAP
      if (fileHandle ≠ −1) {
        munmap(reinterpret_cast⟨char *⟩(dict), flen);
        close(fileHandle);
      }
#else
      if (dict ≠ Λ) {
        delete dict;
      }
#endif
    }
    void describe(ostream &os = cout)
    { }
    static unsigned int itemSize(unsigned char *p)
    {      /* Return item size in bytes */
      return p[0] + 1 + letterCountSize + categoryCountSize;
    }
    unsigned char *first(void)
    {      /* Pointer to first item */
      return dict + 4;      /* Have to skip number of words */
    }
    static unsigned char *next(unsigned char *p)
    {      /* Pointer to item after this one */
```

```
    p += itemSize(p);
    return (*p ≡ 0) ? Λ : p;
  }
  static unsigned char *letterCount(unsigned char *p)
  {     /* Pointer to letter count table for item */
    return p + p[0] + 1;
  }
  static unsigned char *characterCategories(unsigned char *p)
  {     /* Pointer to character category table */
    return letterCount(p) + letterCountSize;
  }
  static unsigned int length(unsigned char *p)
  {     /* Get text length */
    return p[0];
  }
  static void getText(unsigned char *p, string *s)
  {     /* Assign text value to string */
    s→assign(reinterpret_cast⟨char *⟩(p + 1), p[0]);
  }
  static void setDictionaryWordCheap(dictionaryWord *w, unsigned char *p)
  {
    getText(p, &(w→text));
    memcpy(w→letterCount, letterCount(p), letterCountSize);
  }     /* Offsets of fields in the letter category table */
  static const unsigned int C_upper = 0, C_lower = 1, C_digits = 2, C_spaces = 3,
      C_punctuation = 4, C_ISOupper = 5, C_ISOlower = 6, C_ISOpunctuation = 7;
  static void printItem(unsigned char *p, ostream &os = cout);
  void printDictionary(ostream &os = cout);
};
```

22. Since we're going to access the binary dictionary intensively (via the auxiliary dictionary's pointers), we need to load it into memory. The preferred means for accomplishing this is to memory map the file containing the dictionary into our address space and delegate management of it in virtual memory to the system. There's no particular benefit to this if you're running the program sporadically as a single user, but in server applications such as a CGI script driving the program, memory mapping with the `MAP_SHARED` attribute can be a huge win, since the database is shared among all processes requiring it and, once brought into virtual memory, need not be reloaded when subsequent processes require it.

But of course not every system supports memory mapping. Our `autoconf` script detects whether the system supports the *mmap* function and, if not, sets a configuration variable which causes us to fall back on reading the dictionary into a dynamically allocated memory buffer in the process address space.

```
⟨ Bring binary dictionary into memory 22 ⟩ ≡
  FILE *fp;
  fp = fopen(s.c_str( ), "rb");
  if (fp ≡ Λ) {
    cout ≪ "Cannot␣open␣binary␣dictionary␣file␣" ≪ s ≪ endl;
    exit(1);
  }
  fseek(fp, 0, 2);
  flen = ftell(fp);
#ifndef HAVE_MMAP
  dict = new unsigned char[flen];
  rewind(fp);
  fread(dict, flen, 1, fp);
#endif
  fclose(fp);
#ifdef HAVE_MMAP
  fileHandle = open(s.c_str( ), O_RDONLY);
  dict = reinterpret_cast⟨unsigned char *⟩(mmap((caddr_t)0, flen, PROT_READ,
      MAP_SHARED | MAP_NORESERVE, fileHandle, 0));
#endif
  nwords = (((((dict[0] ≪ 8) | dict[1]) ≪ 8) | dict[2]) ≪ 8) | dict[3];
  if (verbose) {
    cerr ≪ "Loaded␣" ≪ nwords ≪ "␣words␣from␣binary␣dictionary␣" ≪ s ≪ "." ≪ endl;
  }
```

This code is used in section 21.

23. *printItem* prints a human readable representation of the item its argument points to on the designated output stream.

```
⟨ Class implementations 8 ⟩ +≡
  void binaryDictionary::printItem(unsigned char *p, ostream &os)
  {
    unsigned int textLen = *p++;
    string text(reinterpret_cast⟨char *⟩(p), textLen);
    os ≪ text ≪ endl;
    p += textLen;
    for (unsigned int i = 0; i < letterCountSize; i++) {
      unsigned int n = *p++;
      if (n > 0) {
        os ≪ "␣␣" ≪ static_cast⟨char⟩('a' + i) ≪ "␣␣" ≪ setw(2) ≪ n ≪ endl;
      }
    }
    unsigned int upper, lower, digits, spaces, punctuation, ISOupper, ISOlower, ISOpunctuation;
    upper = *p++;
    lower = *p++;
    digits = *p++;
    spaces = *p++;
    punctuation = *p++;
    ISOupper = *p++;
    ISOlower = *p++;
    ISOpunctuation = *p;
    os ≪ "␣␣ASCII:␣Letters:␣" ≪ (upper + lower) ≪ "␣␣(Upper:␣" ≪ upper ≪ "␣␣Lower:␣" ≪
        lower ≪ ").␣␣Digits:␣" ≪ digits ≪ "␣␣Punctuation:␣" ≪ punctuation ≪ "␣␣Blanks:␣" ≪
        spaces ≪ endl;
    os ≪ "␣␣ISO:␣Letters:␣" ≪ (ISOupper + ISOlower) ≪ "␣␣(Upper:␣" ≪ ISOupper ≪
        "␣␣Lower:␣" ≪ ISOlower ≪ ").␣␣Punctuation:␣" ≪ ISOpunctuation ≪ endl;
  }
```

24. To print the entire dictionary, we simply iterate over the entries and ask each to print itself with *printItem*.

```
⟨ Class implementations 8 ⟩ +≡
  void binaryDictionary::printDictionary(ostream &os)
  {
    unsigned char *p = first( );
    while (p ≠ Λ) {
      printItem(p, os);
      p = next(p);
    }
  }
```

25. Main program.

The main program is rather simple. We initialise the letter category table, analyse the command line, and then do whatever it asked us to do.

```
⟨ Main program 25 ⟩ ≡
  int main(int argc, char *argv[])
  {
    int opt;
    build_letter_category( );
    ⟨ Process command-line options 56 ⟩;
    ⟨ Verify command-line arguments 59 ⟩;
    ⟨ Perform requested operation 26 ⟩;
    return 0;
  }
```

This code is used in section 6.

26. The global variable *cgi_step* tells us what we're expected to do in this session. If 0, this is a command-line invocation to generate all the anagrams for the target (or just a list of words in anagrams if `--bail` is set). Otherwise, this is one of the three phases of CGI script processing. In the first phase, we generate a table of words appearing in anagrams. In the second, we generate anagrams containing a selected word (or all anagrams, if the HTML template admits that option and the box is checked). In the third, we display all the permutations of words in a selected anagram; for this step the dictionary is not required and is not loaded.

```
⟨ Perform requested operation 26 ⟩ ≡
  if (exportfile ≠ "") {
    ⟨ Build the binary dictionary from a word list 28 ⟩;
  }
  dictionaryWord w(target);
  w.onlyLetters( );
  w.toLower( );
  ⟨ Load the binary dictionary if required 27 ⟩;
  switch (cgi_step) {
  case 0:     /* Non-CGI—invoked from the command line */
    {
      direct_output = true;
      ⟨ Specify target from command line argument if not already specified 29 ⟩;
      if (permute) {
        ⟨ Generate permutations of target phrase 31 ⟩;
      }
      else {
        ⟨ Find anagrams for word 30 ⟩;
      }
    }
    break;
  case 1:     /* Initial request for words in anagrams */
    bail = true;
    ⟨ Find anagrams for word 30 ⟩;
    generateHTML(cout, '2');
    break;
  case 2:     /* Request for anagrams beginning with given word */
    ⟨ Find anagrams for word 30 ⟩;
    generateHTML(cout, '3');
    break;
  case 3:     /* Request for permutations of a given anagram */
    generateHTML(cout);
    break;
  }
```

This code is used in section 25.

27. If the requested operation requires the binary dictionary, load it from the file. After loading the binary dictionary, an auxiliary dictionary consisting of pointers to items in the binary dictionary which can appear in anagrams of the target w is built with *build_auxiliary_dictionary*.

```
⟨ Load the binary dictionary if required 27 ⟩ ≡
  binaryDictionary bdict;
  if ((cgi_step < 3) ∧ (¬permute)) {
    if (bdictfile ≠ "") {
      bdict.loadFromFile(bdictfile);
    }
  }
```

This code is used in section 26.

28. When the `--export` option is specified, we load the word list file given by the `--dictionary` option (or the default), filter and sort it as required, and write it in binary dictionary format. The program immediately terminates after creating the binary dictionary.

```
⟨ Build the binary dictionary from a word list 28 ⟩ ≡
  dictionary dict;
  ofstream os(exportfile.c_str(), ios::binary | ios::out);
  ifstream dif(dictfile.c_str());
  dict.loadFromFile(dif, false, false);
  dict.sortByDescendingLength();
#ifdef DICTECHO
  {
    ofstream es("common.txt");
    for (int i = 0; i < dict.size(); i++) {
      es ≪ dict[i].text ≪ endl;
    }
    es.close();
  }
#endif
  dict.exportToBinaryFile(os);
  os.close();
  return 0;
```

This code is used in section 26.

29. If the user has not explicitly specified the target with the `--target` option, obtain it from the first command line argument, if any.

```
⟨ Specify target from command line argument if not already specified 29 ⟩ ≡
  if ((optind < argc) ∧ (target ≡ "")) {
    target = argv[optind];
    w.set(target);
    w.onlyLetters();
    w.toLower();
    optind++;
  }
```

This code is used in section 26.

30. To find the anagrams for a given word, we walk through the dictionary in a linear fashion and use the relational operators on the letter count tables to test "containment". A dictionary word is *contained* within the target phrase if it has the same number or fewer of each letter in the target, and no letters which do not appear in the target.

```
⟨ Find anagrams for word 30 ⟩ ≡
  build_auxiliary_dictionary(&bdict, &w);
  if (optind < argc) {
    for (int n = optind; n < argc; n++) {
      dictionaryWord *given = new dictionaryWord(argv[n]);
      seed = seed + *given;
      anagram.push_back(given);
    }
    if ((seed ≤ w) ∧ (¬(seed > w))) {
      anagram_search(w, anagram, 0, bail, bail ? 0 : −1);
    }
    else {
      cerr ≪ "Seed␣words␣are␣not␣contained␣in␣target." ≪ endl;
      return 2;
    }
  }
  else if (seed.length( ) > 0) {
    if ((seed ≤ w) ∧ (¬(seed > w))) {
      anagram_search(w, anagram, 0, bail, bail ? 0 : −1);
    }
    else {
      cerr ≪ "Seed␣words␣are␣not␣contained␣in␣target." ≪ endl;
      return 2;
    }
  }
  else {
    for (unsigned int n = 0; n < auxdictl; n++) {
      unsigned char *p = auxdict[n];
      vector⟨dictionaryWord *⟩ anagram;
      dictionaryWord aw;
      binaryDictionary::setDictionaryWordCheap(&aw, p);
      anagram.push_back(&aw);
      anagram_search(w, anagram, n, bail, bail ? 0 : −1);
    }
  }
```

This code is used in section 26.

31. When the `--permute` option is specified, load words from the target phrase into the *permutations* vector and generate and output the permutations.

```
⟨ Generate permutations of target phrase 31 ⟩ ≡
  ⟨ Load words of target into permutations vector 32 ⟩;
  ⟨ Enumerate and print permutations 33 ⟩;
```

This code is used in section 26.

32. Walk through the target string, extract individual (space separated) words, and place them in the *permutations* vector. We're tolerant of extra white space at the start, end, or between words.

```
⟨ Load words of target into permutations vector 32 ⟩ ≡
  bool done = false;
  string::size_type pos = target.find_first_not_of('␣'), spos;
  while (¬done) {
    if ((spos = target.find_first_of('␣', pos)) ≠ string::npos) {
      permutations.push_back(target.substr(pos, spos − pos));
      pos = target.find_first_not_of('␣', spos + 1);
    }
    else {
      if (pos < target.length()) {
        permutations.push_back(target.substr(pos));
      }
      done = true;
    }
  }
```

This code is used in section 31.

33. Generate all of the permutations of the specified words and print each. Note that we must sort *permutations* so that the permutation generation will know when it's done.

```
⟨ Enumerate and print permutations 33 ⟩ ≡
  int n = permutations.size(), nbang = 1;
  vector⟨string⟩::iterator p;
  while (n > 1) {
    nbang *= n−−;
  }
  sort(permutations.begin(), permutations.end());
  do {
    for (p = permutations.begin(); p ≠ permutations.end(); p++) {
      if (p ≠ permutations.begin()) {
        dout ≪ "␣";
      }
      dout ≪ *p;
    }
    dout ≪ endl;
  } while (next_permutation(permutations.begin(), permutations.end()));
```

This code is used in section 31.

34. Here we collect together the assorted global functions of various types.

```
⟨ Global functions 34 ⟩ ≡
  ⟨ Character category table initialisation 66 ⟩;
  ⟨ Auxiliary dictionary construction 36 ⟩;
  ⟨ Anagram search auxiliary function 38 ⟩;
  ⟨ HTML generator 42 ⟩;
```

See also section 55.

This code is used in section 6.

35. Auxiliary Dictionary.

In almost every case we can drastically speed up the search for anagrams by constructing an *auxiliary dictionary* consisting of only those words which can possible occur in anagrams of the target phrase. Words which are longer, or contain more of a given letter than the entire target phrase may be immediately excluded and needn't be tested in the exhaustive search phase.

⟨ Global variables 35 ⟩ ≡

```
unsigned char **auxdict = Λ;
unsigned int auxdictl = 0;
```

See also sections 37, 54, 58, 62, 63, 64, and 65.

This code is cited in section 54.

This code is used in section 6.

36. The process of building the auxiliary dictionary is painfully straightforward, but the key is that we only have to do it *once*. We riffle through the word list (which has already been sorted in descending order by length of word), skipping any words which, by themselves are longer than the *target* and hence can't possibly appear in anagrams of it. Then, we allocate a pointer table as long as the remainder of the dictionary (which often wastes a bunch of space, but it's much faster than using a **vector** or dynamically expanding a buffer as we go) and fill it with pointers to items in the dictionary which are contained within possible anagrams of the *target*. It is this table we'll use when actually searching for anagrams.

```
⟨ Auxiliary dictionary construction 36 ⟩ ≡
  static void build_auxiliary_dictionary(binaryDictionary *bd, dictionaryWord *target)
  {
    if (auxdict ≠ Λ) {
      delete auxdict;
      auxdict = Λ;
      auxdictl = 0;
    }
    unsigned long i;
    unsigned char *p = bd→first( );
    unsigned int tlen = target→length( );      /* Quick reject all entries longer than the target */
    for (i = 0; i < bd→nwords; i++) {
      if (binaryDictionary::length(p) ≤ tlen) {
        break;
      }
      p = binaryDictionary::next(p);
    }     /* Allocate auxiliary dictionary adequate to hold balance */
    if (i < bd→nwords) {
      auxdict = new unsigned char*[bd→nwords − i];
      for ( ; i < bd→nwords; i++) {
        unsigned char *lc = binaryDictionary::letterCount(p);
        for (unsigned int j = 0; j < binaryDictionary::letterCountSize; j++) {
          if (lc[j] > target→letterCount[j]) {
            goto busted;
          }
        }
        auxdict[auxdictl++] = p;
      busted: ;
        p = binaryDictionary::next(p);
      }
    }
  }
```

This code is used in section 34.

37. Anagram Search Engine.

The anagram search engine is implemented by the recursive procedure *anagram_search*. It is called with a **dictionaryWord** set to the target phrase and a vector (initially empty) of pointers to **dictionaryWord**s which are candidates to appear in anagrams—in other words, the sum of the individual letter counts of items in the vector are all less than or equal than those of the target phrase.

On each invocation, it searches the auxiliary dictionary (from which words which cannot possibly appear in anagrams of the target because their own letter counts are not contained in it) and, upon finding a word which, added to the words with which the function was invoked, is still contained within the target, adds the new word to the potential anagram vector and recurses to continue the process. The process ends either when an anagram is found (in which case it is emitted to the designated destination) or when no word in the auxiliary dictionary, added to the words already in the vector, can be part of an anagram of the target. This process continues until the outer-level search reaches the end of the dictionary, at which point all anagrams have been found.

If the *bail* argument is *true*, the stack will be popped and the top level dictionary search resumed with the next dictionary word.

```
⟨ Global variables 35 ⟩ +≡
  vector⟨string⟩ anagrams;
  vector⟨string⟩ firstwords;
  vector⟨string⟩ anagrams_for_word;
  vector⟨string⟩ permutations;
```

38.

```
⟨ Anagram search auxiliary function 38 ⟩ ≡
  static bool anagram_search(dictionaryWord &target, vector⟨dictionaryWord *⟩ &a, unsigned int
        n, bool bail = false, int prune = −1){
```

See also sections 39, 40, and 41.

This code is used in section 34.

39. Assemble the base word from the contained words in the candidate stack. In the interest of efficiency (remember that this function can be called recursively millions of times in a search for anagrams of a long phrase with many high-frequency letters), we explicitly add the letter counts for the words rather than using the methods in **dictionaryWord** which compute character class counts we don't need here.

```
⟨ Anagram search auxiliary function 38 ⟩ +≡
  vector⟨dictionaryWord *⟩::size_type i;
  dictionaryWord wbase(0);
  for (i = 0; i < a.size(); i++) {
    wbase.text += a[i]→text;
    for (unsigned int j = 0; j < binaryDictionary::letterCountSize; j++) {
      wbase.letterCount[j] += a[i]→letterCount[j];
    }
  }
```

40. The first thing we need to determine is if we're *already done*; in other words, is the existing base word itself an anagram. This test is just a simple test for equality of letter counts.

```
⟨ Anagram search auxiliary function 38 ⟩ +≡
  if (memcmp(target.letterCount, wbase.letterCount, binaryDictionary::letterCountSize) ≡ 0) {
    string result;
    for (i = 0; i < a.size(); i++) {
      if (i > 0) {
        result += "␣";
      }
      result += a[i]→text;
      if (static_cast⟨int⟩(i) ≡ prune) {
        break;
      }
    }
    if (direct_output) {
      dout ≪ result ≪ endl;
    }
    else {
      anagrams.push_back(result);
    }
    return true;
  }
```

41. Okay, we're not done. The base word consequently contains fewer of one or more letters than the target. So, we walk forward through the dictionary starting with the current word (the one just added to the base) until the end. For each dictionary word, we test whether the base phrase with that word concatenated at the end remains contained within the target or if it "goes bust" either by containing a letter not in the target or too many of one of the letters we're still looking for.

```
⟨ Anagram search auxiliary function 38 ⟩ +≡
  for ( ; n < auxdictl; n++) {
    unsigned char *p = auxdict[n];
    if ((wbase.length( ) + binaryDictionary::length(p)) ≤ target.length( )) {
      if (target.contained(&wbase, p)) {
        dictionaryWord aw;
        binaryDictionary::setDictionaryWordCheap(&aw, p);
        a.push_back(&aw);
        bool success = anagram_search(target, a, n, bail, prune);
        a.pop_back( );
        if (bail ∧ success) {
          return (a.size( ) > 1);
        }
      }
    }
  }
  return false; }
```

42. HTML Generator.

The HTML generator creates an HTML result page containing the anagrams or permutations requested by the user. It operates by copying a "template" file to the designated output stream, interpolating variable content at locations indicated by markers embedded in the template.

```
⟨ HTML generator 42 ⟩ ≡
  static void generateHTML(ostream &os, char stopAt = 0)
  {
    ifstream is(HTML_template.c_str( ));
    string s;
    bool skipping = false;
    while (getline(is, s)) {
      if (s.substr(0, 5) ≡ "<!--␣") {
        ⟨ Process meta-command in HTML template 43 ⟩;
      }
      else {
        if (¬skipping) {
          os ≪ s ≪ endl;
        }
      }
    }
  }
```

This code is used in section 34.

43. HTML comments beginning in column 1 are used as meta-commands for purposes such as marking sections of the file to be skipped and to trigger the inclusion of variable content.

```
⟨ Process meta-command in HTML template 43 ⟩ ≡
  if (s[5] ≡ '@') {
    ⟨ Process include meta-command 45 ⟩;
  }
  else {
    ⟨ Process section marker meta-command 44 ⟩;
  }
```

This code is used in section 42.

44. As the interaction with the user progresses, we wish to include additional steps in the HTML reply returned. The template file includes section markers between each step in the interaction. If the *stopAt* argument matches the character in the section marker, the balance of the file is skipped until the special "`X`" section marker at the bottom is encountered.

```
⟨ Process section marker meta-command 44 ⟩ ≡
  if (s[5] ≡ 'X') {
    skipping = false;
  }
  else if (s[5] ≡ stopAt) {
    skipping = true;
  }
```

This code is used in section 43.

45. Meta-commands which begin with "`@`" trigger inclusion of variant material in the file, replacing the marker. Naturally, we process these includes only when we aren't skipping sections in the file.

```
⟨ Process include meta-command 45 ⟩ ≡
  if (¬skipping) {
    switch (s[6]) {
    case '1':
      os ≪ "␣␣␣␣value=\"" ≪ target ≪ "\"" ≪ endl;
      break;
    case '2': ⟨ Generate first word selection list 46 ⟩;
      break;
    case '3':
      ⟨ Generate first word hidden argument 47 ⟩;
      break;
    case '4':
      ⟨ Generate list of anagrams containing specified word 48 ⟩;
      break;
    case '5':
      ⟨ Generate all permutations of selected anagram 51 ⟩;
      break;
    case '6':
      ⟨ Generate pass-on of target and firstwords to subsequent step 49 ⟩;
      ⟨ Generate pass-on of anagrams generated in step 2 50 ⟩;
      break;
    }
  }
```

This code is used in section 43.

46. The following code generates the list used to select the set of anagrams to be generated based on a word it contains. In the first phase of processing, this is simply the pruned results from the dictionary search. Subsequently, the list is simply parroted from a `hidden` form field passed in from the previous step.

```
⟨ Generate first word selection list 46 ⟩ ≡
  {
    vector⟨string⟩ &svec = (cgi_step ≡ 1) ? anagrams : firstwords;
    vector⟨string⟩::size_type nfound = svec.size( );
    if (nfound > 12) {
      nfound = 12;
    }
    os ≪ "␣␣␣␣size=" ≪ nfound ≪ ">" ≪ endl;
    for (vector⟨string⟩::iterator p = svec.begin( ); p ≠ svec.end( ); p++) {
      os ≪ "␣␣␣␣<option>" ≪ *p ≪ endl;
    }
  }
```

This code is used in section 45.

47. Polly want a crude hack? Here's where we generate the list of first words which subsequent steps parrot to populate the first word selection box. If we're in a subsequent step this is, in turn parroted from the preceding step.

```
⟨ Generate first word hidden argument 47 ⟩ ≡
  {
    vector⟨string⟩ &svec = (cgi_step ≡ 1) ? anagrams : firstwords;
    os ≪ "<input␣type=\"hidden\"␣name=\"target\"␣value=\"" ≪ target ≪ "\">" ≪ endl;
    os ≪ "<input␣type=\"hidden\"␣name=\"firstwords\"␣value=\"";
    for (vector⟨string⟩::iterator p = svec.begin(); p ≠ svec.end(); p++) {
      if (p ≠ svec.begin()) {
        os ≪ ",";
      }
      os ≪ *p;
    }
    os ≪ "\">" ≪ endl;
  }
```

This code is used in section 45.

48. Here we emit the list of anagrams which contain a word chosen from the list of words in step 2. As with the table of first words above, we either emit these words from the *anagrams* vector if we're in step 2 or from the canned *anagrams_for_word* in subsequent steps.

```
⟨ Generate list of anagrams containing specified word 48 ⟩ ≡
  {
    vector⟨string⟩ &svec = (cgi_step ≡ 2) ? anagrams : anagrams_for_word;
    vector⟨string⟩::size_type nfound = svec.size();
    if (nfound > 12) {
      nfound = 12;
    }
    os ≪ "␣␣␣␣size=" ≪ nfound ≪ ">" ≪ endl;
    for (vector⟨string⟩::iterator p = svec.begin(); p ≠ svec.end(); p++) {
      os ≪ "␣␣␣␣<option>" ≪ *p ≪ endl;
    }
  }
```

This code is used in section 45.

49. In steps subsequent to 1, we need to preserve the target and list of first words selection box. This code generates hidden input fields to pass these values on to subsequent steps.

```
⟨ Generate pass-on of target and firstwords to subsequent step 49 ⟩ ≡
  {
    os ≪ "<input␣type=\"hidden\"␣name=\"target\"␣value=\"" ≪ target ≪ "\">" ≪ endl;
    os ≪ "<input␣type=\"hidden\"␣name=\"firstwords\"␣value=\"";
    for (vector⟨string⟩::iterator p = firstwords.begin(); p ≠ firstwords.end(); p++) {
      if (p ≠ firstwords.begin()) {
        os ≪ ",";
      }
      os ≪ *p;
    }
    os ≪ "\">" ≪ endl;
  }
```

This code is used in section 45.

50. Similarly, in steps 2 and later we need to pass on the anagrams generated by step 2. We also output these as a hidden input field.

```
⟨ Generate pass-on of anagrams generated in step 2 50 ⟩ ≡
  {
    vector⟨string⟩ &svec = (cgi_step ≡ 2) ? anagrams : anagrams_for_word;
    os ≪ "<input␣type=\"hidden\"␣name=\"anagrams\"␣value=\"";
    for (vector⟨string⟩::iterator p = svec.begin(); p ≠ svec.end(); p++) {
      if (p ≠ svec.begin()) {
        os ≪ ",";
      }
      os ≪ *p;
    }
    os ≪ "\">" ≪ endl;
  }
```

This code is used in section 45.

51. When the user selects one of the anagrams presented in step 3 and proceeds to step 4, we prepare a list of all possible permutations of the anagram. The user's gotta be pretty lazy not to be able to to this in his head, but what the heck, *many* Web users are lazy! Besides, STL's *next_permutation* algorithm does all the work, so even a lazy implementor doesn't have to overexert himself.

Before the table of permutations, we emit the `rows=` specification and close the `textarea` tag. For a list of n items there are $n!$ permutations, so we compute the factorial and size the box accordingly.

```
⟨ Generate all permutations of selected anagram 51 ⟩ ≡
  {
    int n = permutations.size(), nbang = 1, cols = 1;
    vector⟨string⟩::iterator p;
    while (n > 1) {
      nbang *= n--;
    }
    for (p = permutations.begin(); p ≠ permutations.end(); p++) {
      if (p ≠ permutations.begin()) {
        cols++;
      }
      cols += p→length();
    }
    cout ≪ "cols=" ≪ cols ≪ "␣rows=" ≪ nbang ≪ ">" ≪ endl;
    sort(permutations.begin(), permutations.end());
    do {
      for (p = permutations.begin(); p ≠ permutations.end(); p++) {
        if (p ≠ permutations.begin()) {
          os ≪ "␣";
        }
        os ≪ *p;
      }
      os ≪ endl;
    } while (next_permutation(permutations.begin(), permutations.end()));
  }
```

This code is used in section 45.

52. The following include files provide access to system and library components.

⟨ System include files 52 ⟩ ≡

```
#include "config.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>
  using namespace std;
#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>
#include <string.h>
#ifdef HAVE_STAT
#include <sys/stat.h>
#endif
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#ifdef WIN32      /* Lazy way to avoid manually modifying config.h for Win32 builds */
#undef HAVE_MMAP
#endif
#ifdef HAVE_MMAP
#include <sys/mman.h>
#endif
#include "getopt.h"      /* Use our own getopt, which supports getopt_long */
  extern"C" void uncgi(void);
    /* We use uncgi to transform CGI arguments into environment variables */
```

This code is used in section 5.

53. Here are the global variables we use to keep track of command line options.

⟨ Command line arguments 53 ⟩ ≡
```
#ifdef NEEDED
  static bool flattenISOchars = false;      /* Flatten ISO 8859-1 8-bit codes to ASCII */
#endif
  static bool bail = false;      /* Bail out on first match for a given word */
  static string dictfile = "crossword.txt";      /* Dictionary file name */
  static string bdictfile = "wordlist.bin";      /* Binary dictionary file */
  static string exportfile = "";      /* Export dictionary file name */
  static string target = "";      /* Target phrase */
  static dictionaryWord seed("");      /* Seed word */
  static dictionaryWord empty("");      /* Empty dictionaryWord */
  static vector<dictionaryWord *> anagram;      /* Anagram search stack */
  static int cgi_step = 0;      /* CGI processing step or 0 if not CGI-invoked */
  static bool cgi_all = false;      /* Generate all anagrams in CGI step 2 */
  static bool html = false;      /* Generate HTML output ? */
  static string HTML_template = "template.html";      /* Template for HTML output */
  static bool direct_output = false;      /* Write anagrams directly to stream ? */
  static ostream &dout = cout;      /* Direct output stream */
  static bool permute = false;      /* Generate permutations of target */
```
This code is used in section 6.

54. The following options are referenced in class definitions and must be placed in the ⟨ Global variables 35 ⟩ section so they'll be declared first.

⟨ Global variables 35 ⟩ +≡
```
  static bool verbose = false;      /* Print verbose processing information */
```

55. Procedure *usage* prints how-to-call information. This serves as a reference for the option processing code which follows. Don't forget to update *usage* when you add an option!

```
⟨ Global functions 34 ⟩ +≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Anagram␣Finder.␣␣Call\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣with␣" ≪ PRODUCT ≪ "␣[options]␣'phrase'␣[contained␣words]\n";
    cout ≪ "\n";
    cout ≪ "Options:\n";
    cout ≪ "␣␣␣␣--all␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Generate␣all␣anagrams␣in␣CGI␣step␣2\n";
    cout ≪ "␣␣␣␣--bail␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bail␣out␣after␣first␣match␣for␣a␣given␣word\n";
    cout ≪ "␣␣␣␣--bindict,␣-b␣file␣␣␣␣␣Use␣file␣as␣binary␣dictionary\n";
    cout ≪ "␣␣␣␣--cgi␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Set␣options␣from␣uncgi␣environment␣variables\n";
    cout ≪ "␣␣␣␣--copyright␣␣␣␣␣␣␣␣␣␣␣␣Print␣copyright␣information\n";
    cout ≪ "␣␣␣␣--dictionary,␣-d␣file␣␣Use␣file␣as␣dictionary\n";
    cout ≪ "␣␣␣␣--export␣file␣␣␣␣␣␣␣␣␣␣Export␣binary␣dictionary␣file\n";
#ifdef NEEDED
    cout ≪ "␣␣␣␣--flatten-iso␣␣␣␣␣␣␣␣␣␣Flatten␣ISO␣8859-1␣8-bit␣codes␣to␣ASCII\n";
#endif
    cout ≪ "␣␣␣␣--help,␣-u␣␣␣␣␣␣␣␣␣␣␣␣␣Print␣this␣message\n";
    cout ≪ "␣␣␣␣--html␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Generate␣HTML␣output\n";
    cout ≪ "␣␣␣␣--permute,␣-p␣␣␣␣␣␣␣␣␣␣Generate␣permutations␣of␣target\n";
    cout ≪ "␣␣␣␣--seed,␣-s␣word␣␣␣␣␣␣␣␣Specify␣seed␣word\n";
    cout ≪ "␣␣␣␣--step␣number␣␣␣␣␣␣␣␣␣␣CGI␣processing␣step\n";
    cout ≪ "␣␣␣␣--target,␣-t␣phrase␣␣␣␣␣Target␣phrase\n";
    cout ≪ "␣␣␣␣--template␣␣␣␣␣␣␣␣␣␣␣␣␣Template␣for␣HTML␣output\n";
    cout ≪ "␣␣␣␣--verbose,␣-v␣␣␣␣␣␣␣␣␣␣Print␣processing␣information\n";
    cout ≪ "␣␣␣␣--version␣␣␣␣␣␣␣␣␣␣␣␣␣␣Print␣version␣number\n";
    cout ≪ "\n";
    cout ≪ "by␣John␣Walker\n";
    cout ≪ "http://www.fourmilab.ch/\n";
  }
```

56. We use *getopt_long* to process command line options. This permits aggregation of single letter options without arguments and both `-d`*arg* and `-d` *arg* syntax. Long options, preceded by `--`, are provided as alternatives for all single letter options and are used exclusively for less frequently used facilities.

```
⟨ Process command-line options 56 ⟩ ≡
  static const struct option long_options[] = {
  {"all", 0, Λ, 206},
  {"bail", 0, Λ, 208},
  {"bindict", 1, Λ, 'b'},
  {"cgi", 0, Λ, 202},
  {"copyright", 0, Λ, 200},
  {"dictionary", 1, Λ, 'd'},
  {"export", 1, Λ, 207},
#ifdef NEEDED
  {"flatten-iso", 0, Λ, 212},
#endif
  {"html", 0, Λ, 203},
  {"permute", 0, Λ, 'p'},
  {"seed", 1, Λ, 's'},
  {"step", 1, Λ, 205},
  {"target", 1, Λ, 't'},
  {"template", 1, Λ, 204},
  {"help", 0, Λ, 'u'},
  {"verbose", 0, Λ, 'v'},
  {"version", 0, Λ, 201},
  {0, 0, 0, 0}
  };
  int option_index = 0;
  while ((opt = getopt_long(argc, argv, "b:d:ps:t:uv", long_options, &option_index)) ≠ −1) {
    switch (opt) {
    case 206:    /* --all Generate all anagrams in CGI step 2 */
      cgi_all = true;
      break;
    case 208:    /* --bail Bail out after first match for word */
      bail = true;
      break;
    case 'b':    /* -b, --bindict binaryDictionary */
      bdictfile = optarg;
      break;
    case 202:    /* --cgi Set options from uncgi environment variables */
      ⟨ Set options from uncgi environment variables 57 ⟩;
      break;
    case 200:    /* --copyright Print copyright information */
      cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
      return 0;
    case 'd':    /* -d, --dictionary dictionary-file */
      dictfile = optarg;
      break;
    case 207:    /* --export dictionary-file */
      exportfile = optarg;
      break;
#ifdef NEEDED
    case 212:    /* --flatten-iso Flatten ISO 8859-1 8-bit codes to ASCII */
```

```
    flattenISOchars = true;
    break;
#endif
  case 203:    /* --html Generate HTML output */
    html = true;
    break;
  case 'p':    /* -p, --permute Generate permutations of target */
    permute = true;
    break;
  case 's':    /* -s, --seed seed-word */
    {
      dictionaryWord *given = new dictionaryWord(optarg);
      seed = seed + *given;
      anagram.push_back(given);
    }
    break;
  case 205:    /* –step step-number */
    cgi_step = atoi(optarg);
    break;
  case 't':    /* -t, --target target-phrase */
    target = optarg;
    break;
  case 204:    /* --template HTML-template-file */
    HTML_template = optarg;
    break;
  case 'u':    /* -u, --help Print how-to-call information */
    case '?': usage();
    return 0;
  case 'v':    /* -v, --verbose Print processing information */
    verbose = true;
    break;
  case 201:    /* --version Print version information */
    cout << PRODUCT " " VERSION "\n";
    cout << "Last revised: " REVDATE "\n";
    cout << "The latest version is always available\n";
    cout << "at http://www.fourmilab.ch/anagram/\n";
    cout << "Please report bugs to bugs@fourmilab.ch\n";
    return 0;
  default:
    cerr << "***Internal error: unhandled case " << opt << " in option processing.\n";
    return 1;
  }
}
```

This code is used in section 25.

57. When we're invoked as a CGI program by a Web browser, the `uncgi` program parses the `GET` or `POST` form arguments from the invoking page and sets environment variables for each field in the form. When the `--cgi` option is specified on the command line, this code checks for those variables and sets the corresponding options. This avoids the need for a clumsy and inefficient shell script to translate the environment variables into command line options.

⟨ Set options from uncgi environment variables 57 ⟩ ≡

```
{
  char *env;
  uncgi( );
  if ((env = getenv("WWW_target")) ≠ Λ) {
    target = env;
  }
  if ((env = getenv("WWW_step")) ≠ Λ) {
    cgi_step = atoi(env);
  }
  else {
    cgi_step = 1;
  }
  if (getenv("WWW_all") ≠ Λ) {
    cgi_all = true;
  }
  if ((env = getenv("WWW_firstwords")) ≠ Λ) {
    bool done = false;
    char *endw;
    while (¬done) {
      if ((endw = strchr(env, ',')) ≠ Λ) {
        *endw = 0;
        firstwords.push_back(env);
        env = endw + 1;
      }
      else {
        firstwords.push_back(env);
        done = true;
      }
    }
  }
  if ((env = getenv("WWW_anagrams")) ≠ Λ) {
    bool done = false;
    char *endw;
    while (¬done) {
      if ((endw = strchr(env, ',')) ≠ Λ) {
        *endw = 0;
        anagrams_for_word.push_back(env);
        env = endw + 1;
      }
      else {
        anagrams_for_word.push_back(env);
        done = true;
      }
    }
  }
```

```
  if ((env = getenv("WWW_results")) ≠ Λ) {
    bool done = false;
    char *endw;
    while (¬done) {
      if ((endw = strchr(env, '␣')) ≠ Λ) {
        *endw = 0;
        permutations.push_back(env);
        env = endw + 1;
      }
      else {
        permutations.push_back(env);
        done = true;
      }
    }
  }
  if ((env = getenv("WWW_word")) ≠ Λ) {
    if (¬cgi_all) {
      dictionaryWord *given = new dictionaryWord(env);
      seed = seed + *given;
      anagram.push_back(given);
    }
  }
}
```

This code is used in section 56.

58. Some more global variables to keep track of file name arguments on the command line. . ..

```
⟨ Global variables 35 ⟩ +≡
  static string infile = "-",      /* "-" means standard input or output */
  outfile = "-";
```

59.

```
⟨ Verify command-line arguments 59 ⟩ ≡
  if ((exportfile ≡ "") ∧ (target ≡ "") ∧ (optind ≥ argc)) {
    cerr ≪ "No␣target␣phrase␣specified." ≪ endl ≪
        "Specify␣the␣--help␣option␣for␣how-to-call␣information." ≪ endl;
    return 2;
  }
```

This code is used in section 25.

60. Character set definitions and translation tables.

The following sections define the character set used in the program and provide translation tables among various representations used in formats we emit.

61. ISO 8859-1 special characters.

We use the following definitions where ISO 8859-1 characters are required as strings in the program. Most modern compilers have no difficulty with such characters embedded in string literals, but it's surprisingly difficult to arrange for Plain TEX (as opposed to LATEX) to render them correctly. Since CWEB produces Plain TEX, the path of least resistance is to use escapes for these characters, which also guarantess the generated documentation will work on even the most basic TEX installation. Characters are given their Unicode names with spaces and hyphens replaced by underscores. Character defined with single quotes as **char** have named beginning with `C_`.

```
#define REGISTERED_SIGN "\xAE"
#define C_LEFT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK #AB
#define C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK #BB
#define RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK "\xBB"
```

62. Flat 7-bit ASCII approximation of ISO characters.

The following table is indexed by ISO codes 160 to 255, (#A0–#FF) and gives the flat ASCII rendering of each ISO character. For accented characters, these are simply the characters with the accents removed; for more esoteric characters the translations may be rather eccentric.

```
⟨ Global variables 35 ⟩ +≡        /* Latin 1/Unicode Hex Description */
  static const char *const flattenISO[] = {"␣",       /* #A0 Non-breaking space */
  "!",      /* #A1 Spanish open exclamation */
  "cents",      /* #A2 Cent sign */
  "GBP",      /* #A3 Pounds Sterling */
  "$",      /* #A4 Universal currency symbol */
  "JPY",      /* #A5 Japanese Yen */
  "|",      /* #A6 Broken vertical bar */
  "Sec.",      /* #A7 Section sign */
  "''",      /* #A8 diaeresis */
  "(C)",      /* #A9 Copyright */
  "a",      /* #AA Spanish feminine ordinal indicator */
  "<<",      /* #AB Left pointing guillemet */
  "NOT",      /* #AC Logical not */
  "",      /* #AD Soft (discretionary) hyphen */
  "(R)",      /* #AE Registered trademark */
  "-",      /* #AF Overbar */
  "o",      /* #B0 Degree sign */
  "+/-",      /* #B1 Plus or minus */
  "^2",      /* #B2 Superscript 2 */
  "^3",      /* #B3 Superscript 3 */
  "'",      /* #B4 Acute accent */
  "mu",      /* #B5 Micro sign */
  "PP.",      /* #B6 Paragraph sign */
  ".",      /* #B7 Middle dot */
  ",",      /* #B8 Spacing cedilla */
  "^1",      /* #B9 Superscript 1 */
  "o",      /* #BA Spanish masculine ordinal indicator */
  ">>",      /* #BB Right pointing guillemet */
  "1/4",      /* #BC Fraction one quarter */
  "1/2",      /* #BD Fraction one half */
  "3/4",      /* #BE Fraction three quarters */
  "?",      /* #BF Spanish open question */
  "A",      /* #C0 Accented capital A grave */
  "A",      /* #C1 acute */
  "A",      /* #C2 circumflex */
  "A",      /* #C3 tilde */
  "A",      /* #C4 diaeresis */
  "A",      /* #C5 Capital A ring / Angstrom symbol */
  "Ae",      /* #C6 Capital Ae */
  "C",      /* #C7 Capital C cedilla */
  "E",      /* #C8 Accented capital E grave */
  "E",      /* #C9 acute */
  "E",      /* #CA circumflex */
  "E",      /* #CB diaeresis */
  "I",      /* #CC Accented capital I grave */
  "I",      /* #CD acute */
  "I",      /* #CE circumflex */
```

```
"I",     /* #CF diaeresis */
"Th",     /* #D0 Capital Eth */
"N",     /* #D1 Capital N tilde */
"O",     /* #D2 Accented capital O grave */
"O",     /* #D3 acute */
"O",     /* #D4 circumflex */
"O",     /* #D5 tilde */
"O",     /* #D6 diaeresis */
"x",     /* #D7 Multiplication sign */
"O",     /* #D8 Capital O slash */
"U",     /* #D9 Accented capital U grave */
"U",     /* #DA acute */
"U",     /* #DB circumflex */
"U",     /* #DC diaeresis */
"Y",     /* #DD Capital Y acute */
"Th",     /* #DE Capital thorn */
"ss",     /* #DF German small sharp s */
"a",     /* #E0 Accented small a grave */
"a",     /* #E1 acute */
"a",     /* #E2 circumflex */
"a",     /* #E3 tilde */
"a",     /* #E4 diaeresis */
"a",     /* #E5 Small a ring */
"ae",     /* #E6 Small ae */
"c",     /* #E7 Small c cedilla */
"e",     /* #E8 Accented small e grave */
"e",     /* #E9 acute */
"e",     /* #EA circumflex */
"e",     /* #EB diaeresis */
"i",     /* #EC Accented small i grave */
"i",     /* #ED acute */
"i",     /* #EE circumflex */
"i",     /* #EF diaeresis */
"th",     /* #F0 Small eth */
"n",     /* #F1 Small n tilde */
"o",     /* #F2 Accented small o grave */
"o",     /* #F3 acute */
"o",     /* #F4 circumflex */
"o",     /* #F5 tilde */
"o",     /* #F6 diaeresis */
"/",     /* #F7 Division sign */
"o",     /* #F8 Small o slash */
"u",     /* #F9 Accented small u grave */
"u",     /* #FA acute */
"u",     /* #FB circumflex */
"u",     /* #FC diaeresis */
"y",     /* #FD Small y acute */
"th",     /* #FE Small thorn */
"y"     /* #FF Small y diaeresis */
};
```

63. ISO 8859-1 character types.
The following definitions provide equivalents for `ctype.h` macros which work for ISO-8859 8 bit characters. They require that `ctype.h` be included before they're used.

⟨ Global variables 35 ⟩ +≡
#**define** *isISOspace*(*x*) (*isascii*(((**unsigned char**)(*x*))) ∧ *isspace*(((**unsigned char**)(*x*))))
#**define** *isISOalpha*(*x*)
((*isoalpha*[(((**unsigned char**)(*x*)))/8] & (#80 ≫ ((((**unsigned char**)(*x*))) % 8))) ≠ 0)
#**define** *isISOupper*(*x*)
((*isoupper*[(((**unsigned char**)(*x*)))/8] & (#80 ≫ ((((**unsigned char**)(*x*))) % 8))) ≠ 0)
#**define** *isISOlower*(*x*)
((*isolower*[(((**unsigned char**)(*x*)))/8] & (#80 ≫ ((((**unsigned char**)(*x*))) % 8))) ≠ 0)
#**define** *toISOupper*(*x*) (*isISOlower*(*x*) ? (*isascii*(((**unsigned char**)(*x*))) ? *toupper*(*x*) : (((((**unsigned char**)(*x*)) ≠ #DF) ∧ (((**unsigned char**)(*x*)) ≠ #FF)) ? (((**unsigned char**)(*x*)) − #20) : (*x*))) : (*x*))
#**define** *toISOlower*(*x*)
(*isISOupper*(*x*) ? (*isascii*(((**unsigned char**)(*x*))) ? *tolower*(*x*) : (((**unsigned char**)(*x*)) + #20)) : (*x*))

64. The following tables are bit vectors which define membership in the character classes tested for by the preceding macros.

⟨ Global variables 35 ⟩ +≡
unsigned char *isoalpha*[32] = {0, 0, 0, 0, 0, 0, 0, 0, 127, 255, 255, 224, 127, 255, 255, 224, 0, 0, 0, 0, 0, 0, 0, 0, 255, 255, 254, 255, 255, 255, 254, 255};
unsigned char *isoupper*[32] = {0, 0, 0, 0, 0, 0, 0, 0, 127, 255, 255, 224, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 255, 254, 254, 0, 0, 0, 0};
unsigned char *isolower*[32] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 127, 255, 255, 224, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 255, 255, 254, 255};

65. Character category table.

The character category table allows us to quickly count the number of character by category (alphabetic, upper or lower case, digit, etc.) while scanning a string. Instead of a large number of procedural tests in the inner loop, we use the *build_letter_category* function to initialise a 256 element table of pointers to counters in which the category counts are kept. The table is simply indexed by each character in the string and the indicated category count incremented.

```
#define G_clear()  G_lower = G_upper = G_digits = G_spaces = G_punctuation = G_ISOlower =
        G_ISOupper = G_ISOpunctuation = G_other = 0
```

⟨ Global variables 35 ⟩ +≡

```
unsigned int *letter_category[256];
static unsigned int G_lower, G_upper, G_digits, G_spaces, G_punctuation, G_ISOlower, G_ISOupper,
    G_ISOpunctuation, G_other;
```

66.

```
⟨ Character category table initialisation 66 ⟩ ≡
  static void build_letter_category(void)
  {
    for (int c = 0; c < 256; c++) {
      if (isalpha(c)) {
        if (islower(c)) {
          letter_category[c] = &G_lower;
        }
        else {
          letter_category[c] = &G_upper;
        }
      }
      else {
        if (isdigit(c)) {
          letter_category[c] = &G_digits;
        }
        else if (isspace(c) ∨ (c ≡ '\xA0')) {      /* ISO nonbreaking space is counted as space */
          letter_category[c] = &G_spaces;
        }
        else if (ispunct(c)) {
          letter_category[c] = &G_punctuation;
        }
        else if (isISOalpha(c)) {
          if (isISOlower(c)) {
            letter_category[c] = &G_ISOlower;
          }
          else {
            letter_category[c] = &G_ISOupper;
          }
        }
        else if (c ≥ #A0) {
          letter_category[c] = &G_ISOpunctuation;
        }
        else {
          letter_category[c] = &G_other;
        }
      }
    }
  }
```

This code is used in section 34.

67. Release history.

Release 1: March 2002

Initial release.

Release 1.1: February 2003

Minor source code changes for compatibility with `gcc` 3.2.2. Compatibility with earlier compiler versions is maintained.

Release 1.2: October 2003

Fixes for runaway purity of essence in GCC's library handling of the humble `errno` variable.

68. Development log.

2002 February 23

Created development tree and commenced implementation.

2002 February 26

Much `autoconf` plumbing today to get things to work on Solaris without installing GNU C++ dynamic libraries. I finally ended up having `autoconf` sense the operating system type with `uname` and, if it's "`SunOS`", tack `-static` onto the link. Static linking on Solaris resulted in an error about doubly defined exported symbols in `getopt.c`, but commenting out these symbols seems to fix the problem on Solaris and still work on Linux. Static linking on Linux works fine but takes *forever*, so it's worth making the test to speed up the development cycle.

2002 March 2

Okay, it's basically working now, so it's time to start optimising this horrifically slow program so it doesn't kill the server when we announce it to the public. I started with a test string which, when compiled with `-g -O2` on Lysander, ran 8.690 seconds.

I compiled with `-pg` and ran `gprof` and discovered, to nobody's amazement, that it's spending a huge amount of time counting letters and classifying characters. It was high time to get rid of all that procedural code in the inner loop, so I defined a *letter_category* table which is initialised once with pointers to global category counts with names like *G_lower* and *G_digits*. You can reset the counts with the macro *G_clear*( ). The table is filled in with pointers using procedural code as before. Once the table is built, the category counter simply increments the counter pointed to by indexing the table. The counts are then copied to the corresponding fields in the **dictionaryWord** object. Result: run time fell to 7.030 seconds.

Adding two dictionary words to concatenate them performed a complete re-count of the string. I added a $*+$ operator which simply concatenates the two strings and adds the letter and character category counts. Using this precisely once, where *anagram_search* generates its candidate phrases, reduced run time to 6.510 seconds.

I further sped up candidate testing in *anagram_search* by adding a special-purpose *contained* method to **dictionaryWord**. This takes pointers to two *dictionaryWords* and tests whether the two, considered logically concatenated, are "contained" within the object word. This allows testing containment without ever constructing a new **dictionaryWord** or re-counting letters. This sped up the test case to 3.180 seconds.

Yaaar! Knuth was *really* right when he said that 'premature optimisation is the root of all evil." I was guilty of that in *anagram_search* where I made a test on the combined length of the base phrase and candidate word, without realising thay the *contained* method would reject the candidate in less time than that expended in checking the length.

2002 March 3

Oops. . .specifying a seed or other **dictionaryWord** argument on the command line or via CGI environment variables crashed since the *letter_category* table hadn't been initialised. I moved the initialisation to before the arguments are parsed.

2002 March 5

I obtained a massive speed-up (down to about 0.17 seconds on the test which ran 8.69 seconds at the start of the optimisation) through the expedient of making an initial pass through the dictionary and preparing an "auxiliary dictionary" which contains only words which can possibly appear in anagrams of the target phrase. If a word in the dictionary contains more of any letter than the target phrase, it is excluded, and

we never have to consider it during the expensive recursive process of searching for anagrams. The auxiliary dictionary *auxdict* is a table of pointers to words in the binary dictionary, so referencing through these pointers never requires copiying data.

In the same optimisation pass, I modified the anagram search function which uses the auxiliary dictionary, *a_anagram_search*, to keep its anagram candidate stack as a **vector**⟨**dictionaryWord** ∗⟩ rather than **dictionaryWord**. This avoids copying the object as words are pushed and popped off the stack.

2002 March 12

Added logic to `configure.in` to test whether the system supports memory mapping of files (using the presence of the *mmap* function as a proxy). If it doesn't the **binaryDictionary** *loadFromFile* method allocates an in-memory buffer and reads the binary dictionary into it. We prefer memory mapping since the `MAP_SHARED` attribute allows any number of processes to share the dictionary in page space, which reduces memory requirements and speeds things up enormously in server applications such as CGI scripts.

Integrated the embedded build of `ctangle` and `cweave` in the local `cweb` directory from `EGGSHELL`. Now the `CWEB` tools will automatically be re-built on the user's system.

2002 March 13

To simplify use of the stand-alone program from the command line, I rewrote command line parsing to permit specifying the target as the first command line argument as long as no `--target` options has previously specified it. You may still specify seed words after the target argument regardless of whether it was supplied by `target` or as an unqualified argument.

Switched the `-b` option from a synonym for `--bail` to a synonym of `--bindict`—typical command line users are more likely to specify an alternative dictionary than request single-word bailout.

Added some diagnostic output to binary dictionary creation and loading when the `--verbose` option is specified.

Added a `--permute` (or `-p`) option which generates permutations of the target phrase rather than anagrams. This capability was already in the CGI step 3 processing, but this makes it available to command line users as well.

2002 March 16

Several of the single-letter option abbreviations which take an argument lacked the requisite "`:`" after the letter in the argument to *getopt_long*, resulting in a segmentation fault if any were used. Fixed.

Added documentation to `Makefile.in` and `INSTALL` that this a nerdy user-level application which isn't intended to be installed system-wide.

2002 March 17

Integrated `uncgi` 1.10 as a built-in function with the `-DLIBRARY` option, eliminating the need to install the stand-alone program and the inefficiency executing it entails.

2002 March 20

Updated the `man` page, `anagram.1`, synchronising it with the HTML and built-in program documentation. Changed the version number to 1.0 in anticipation of release.

2002 March 21

Cleaned up building on Win32 with `DJgpp`. The complete program is built by the batch file `makew32.bat`. I added a new `winarch` target to `Makefile.in` to create the `winarch.zip` file containing everything you need to build the Win32 executable.

2002 March 22

Cleaned up all warnings on a `-Wall` build with `gcc`.

Removed unnecessary header file includes.

Removed `CLEAN_BUT_SLOW` code in **dictionaryWord**; we're never going to go back to it, so why not avoid confusion.

Added high level functional documentation for *anagram_search* which is, after all, where all the real work gets done.

Version 1.0.

2003 February 15

Cleaned up in order to compile with `gcc` 3.2.2.

Changed four instances where a function returning an iterator to a string was assigned to a **char** $*$ to assign to a **string** :: *iterator* instead.

Removed "`.h`" from three C++ header file includes.

With these changes, it still compiles on 2.96 without any problems.

Version 1.1.

2003 October 5

The `gcc` thought police have now turned their attention to the humble *errno* variable, necessitating checks in the `configure.in` file for the presence of the `errno.h` header file and *strerror* function which, if present, is used instead of the reference to *sys_errlist*[*errno*] which has worked perfectly for decades. I retested building on `gcc` 3.2.2 and 2.96 on Linux and 2.95.3 on Solaris.

Version 1.2.

69. Index. The following is a cross-reference table for `anagram`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

ANAGRAM