18

18 Bits and pieces

Although one generally prefers to think of data elements as "long integers" or "doubles" or "characters", in the machine they are all just bit patterns packed into successive bytes of memory. Once in a while, that is exactly how you want to think about data.

Usually, you only get to play with bit patterns when you are writing "low-level" code that directly controls input/output devices or performs other operating system services. But there are a few situations where bits get used as parts of high level structures. Two are illustrated in sections 2 and 3 of this chapter. The first section summarizes the facilities that C++ provides for bit manipulations.

Section 18.4 covers another of the more obscure "read only" features of C++. This is the ability to cut up a (long integer) data word into a number of pieces each comprising several bits.

18.1 BIT MANIPULATIONS

Use unsigned longs to store bit patterns

If you need to work with bits, you need a data type to store them in. Generally, unsigned longs are the most convenient of the built in data types. An unsigned long will typically hold 32 bits. (There should be a header file specifying the number of bits in an unsigned long, but this isn't quite standardized among all environments.) If you need bit records with more than 32 bits you will have to use an array of unsigned longs and arrange that your code picks the correct array element to test for a required bit. If you need less than 32 bits, you might chose to use unsigned shorts or unsigned chars.

Unsigned types should be used. Otherwise certain operations may result in '1' bits being added to the left end of your data.

Input and output and constants

You can't actually define bit patterns as literal strings of 0s and 1s, nor can you have them input or output in this form. You wouldn't want to; after, all they are going to be things like

```
01001110010110010111011001111001
00101001101110010101110100000111
```

Instead, hexadecimal notation is used (C and C++ offer octal as an alternative, no one still uses octal so there is no point your learning that scheme). The hexadecimal (hex) scheme uses the hex digits '0', '1', … '9', 'a', 'b', 'c', 'd', 'e', and 'f' to represent groups of four bits:

Hexadecimal symbols for bit patterns

hex	bits	hex	bits
0	0000	1	0001
2	0010	3	0011
4	0100	5	0101
6	0110	7	0111
8	1000	9	1001
a	1010	b	1011
c	1100	d	1101
e	1110	f	1111

(The characters 'A' … 'F' can be used instead of 'a' … 'f'.) Two hex digits encode the bits in an unsigned byte, eight hex digits make up an unsigned long. If you define a constant for say an unsigned long and only give five hex digits, these are interpreted as being the five right most digits with three hex 0s added at the left.

Hexadecimal constants are defined with a leading 0x (or 0X) so that the compiler gets warned that the following digits are to be interpreted as hex rather than decimal:

```
typedef unsigned long Bits;

const Bits kBITS1 = 0x8e58401f;
```

The iostream functions handle hex happily:

```
#include <iostream.h>

typedef unsigned long Bits;

void main()
{
    Bits b1 = 0x1ef2;
    cout.setf(ios::showbase);
    cout << hex << b1 << endl;

    Bits b2;
    cin >> b2;
    cout << b2 << endl;

}
```

When entering a hex number you must start with 0x; so an input for that program could be 0xa2. You should set the "`ios::showbase`" flag on the output stream otherwise the

numbers don't get printed with the leading 0x; it is confusing to see output like 20 when it really is 0x20 (i.e. the value thirty two).

What can you do with bit patterns?

You have all the standard logical operations that combine true/false values. They are illustrated here for groups of four bits (both binary and equivalent hex forms are shown):

```
0110        - (complement, or "not")     ->            1001            complement
0x6                                                   0x9
```

This would be coded using the `~` ("bitwise Not operator"):

```
result = ~value;
```

```
0110        and     1010                 ->            0010            and
0x6                 0xa                               0x2
```

This would be coded using `&` ("bitwise And operator"):

```
result = val1 & val2;
```

(Ouch. We have just met `&` as the "get the address of" operator. Now it has decided to be the "bitwise And operator". It actually has both jobs. You simply have to be careful when reading code to see what its meaning is. Both these meanings are quite different from the logical and operator, `&&`, that was introduced in Part II.)

```
0110        or      1010                 ->            1110            or
0x6                 0xa                               0xe
```

This would be coded using `|` ("bitwise Or operator"):

```
result = val1 | val2;
```

We have had many examples with the `|` operator being used to make up bit patterns used as arguments, e.g. the calls to `open()` that had specifiers like `ios::in | ios::out`. The enumerators `ios::in` and `ios::out` are both values that have one bit encoded; if you want both bits, for an input-output file, you or them to get a result with both bits set.

```
0110        xor     1010                 ->            1100            exclusive or
0x6                 0xa                               0xc
```

This would be coded using `^` ("bitwise Xor operator"):

```
result = val1 ^ val2;
```

(The "exclusive or" of two bit patterns has a one bit where either one or the other but not both of its two inputs had a 1 bit.)

You should test out these basic bit manipulations with the following program:

```
#include <iostream.h>

typedef unsigned long Bits;

int main()
{
    Bits val1, val2, val, result;
    cout.setf(ios::showbase);
    cout.setf(ios::hex,ios::basefield);

    cout << "Enter bit pattern to be complemented : ";
    cin >> val;
    result = ~val;
    cout << "Val : " << val << ", Not val : " << result <<
endl;

    cout << "Enter two bit patterns to be Anded : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 & val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
                        << ", And gives " << result << endl;

    cout << "Enter two bit patterns to be Ored : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 | val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
                        << ", Or gives " << result << endl;

    cout << "Enter two bit patterns to be Xored : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 ^ val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
                        << ", Xor gives " << result << endl;

    return 0;
}
```

Try to work out in advance the hex pattern that you expect as a result for the inputs that you choose. Remember that an input value gets filled out on the left with 0 hex digits so an input of `0x6` will become `0x00000006` and hence when complemented will give `0xfffffff9`.

Abbreviated forms

Of course there are abbreviated forms. If you are updating a bit pattern by combining it with a few more bits, you can use the following:

```
result &= morebits;      // i.e result = result & morebits;
result |= morebits;
result ^= morebits;
```

Moving the bits around

In addition to the bitwise Ands, Ors etc, there are also operators for moving the bits around, or at least for moving all the bits to the left or to the right.

These bit movements are done with the shift operators. They move the bits in a data element left or right by the specified number of places. A left shift operator moves the bits leftwards, adding 0s at the right hand side. A right shift operator moves bits rightwards. This is the one you need to be careful with regarding the use of unsigned values. The right shift operator adds 0s at the left if a data type is unsigned; but if it is given a signed value (e.g. just an ordinary long) it duplicates the leftmost bit when shifting the other bits to the right. This "sign extension" is not usually the behaviour that you want when working with bit patterns. Bits "falling out" at the left or right end of a bit pattern are lost.

Shift operators

The shift operators are:

```
<<  left shift
>>  right shift
```

Again ouch. You know these as the "takes from" and "gives to" operators that work with iostreams.

Overloaded operators

In C++, many things get given more than one job (or, to put it another way, they get "overloaded"). We've just seen that `&` has sometimes got to go and find an address and sometimes it has to combine bit patterns. The job at hand is determined by the context. It is just the same for the `>>` and `<<` operators. If a `>>` operator is located between an input stream variable and some other variable it means "gives to", but if a `>>` operator is between two integer values it means arrange for a right shift operation.

The following code fragment illustrates the working of the shift operators:

```
#include <iostream.h>

typedef unsigned long Bits;

int main()
{
    Bits val, result;
    int i;
    cout.setf(ios::showbase);
    cout.setf(ios::hex,ios::basefield);
    cout << "Enter val "; cin >> val;
    cout << "Enter shift amount "; cin >>i;

    result = val << i;
    cout << "Left shifting gives " << result << endl;
```

```
    result = val >> i;
    cout << "Right shifting gives " << result << endl;

    return 0;
}
```

An example output is:

```
Enter val 0x01234567
Enter shift amount 8
Left shifting gives 0x23456700
Right shifting gives 0x12345
```

Moving by 8 places as requested in the example means that two hex digits (4-bits each) are lost (replaced by zeros).

While you probably find no difficulty in understanding the actual shift operations, you may be doubting whether they have any useful applications. They do; some uses are illustrated in the examples presented in the next two sections.

18.2 MAKING A HASH OF IT

It is common to need to generate a "key value" that summarizes or characterises a complex data object. If data objects are in some sense "equal", then their generated key values must be equal. A key generation process need not be perfect; it is acceptable for two unequal data objects to have the same key value, but ideally the chance of this happening should be very low. A key value can be a (32-bit) unsigned long integer, though preferably it is something larger (e.g. a 64 bit "long long" if your compiler supports such things). The examples here will use unsigned longs to represent such keys.

Where might such keys be needed? There are at least two common applications:

1. As a filter to improve performance when searching a collection for matching data
2. As a summary signature of some data that can be used to check that these data are unchanged since the summary was generated.

The idea of the search filter is that you use the key to eliminate most of the data in a collection, selecting just those data elements that have equal keys. These data elements can then be checked individually, using more elaborate comparisons to find an exact match.

Hashing

The process of generating a key is known as "hashing", and it is something of an art form (i.e. there aren't any universal scientific principles that can lead you to a good hashing function for a specific type of data). Here we consider only the case of text strings because these are the most common data to which hashing is applied

18.2.1 Example hashing function for a character string

If you have a string that you want to summarize in a key, then that key should depend on every character in the string. You can achieve this by any algorithm that mixes up (or "smashes together" of "hashes") the bits from the individual characters of the string.

The "XOR" function is a good way to combine bits because it depends equally on each of its two inputs. If we want a bit pattern that combines bits from all the characters in a string we need a loop that xors the next character into a key, then moves this key left a little to fill up a long integer.

Of course, when you move the key left, some bits fall out the left end. These are the bits that encode the first few characters. If you had a long string, the result could end up depending only on the last few characters in the string.

That problem can be avoided by saving the bits that "fall out the left end" and feeding them back in on the right, xoring them with the new character data.

The following function implements this string hashing:

```
Bits HashString(const char str[])
{
    Bits   Result = 0;
    int    n = strlen(str);
    Bits   Top5Bits = 0xf8000000;
    Bits   Carry = 0x0;
    const int kleftmove = 5;
    const int krightmove = 27;
    for(int i = 0; i< n; i++) {
           Carry = Result & Top5Bits;
           Carry = Carry >> krightmove;
           Result = Result << kleftmove;
           Result ^= Carry;
           Result ^= str[i];
           }
    return Result;
}
```

The statements:

```
Carry = Result & Top5Bits;
Carry = Carry >> krightmove;
```

get the bits occupying the left most five bits of the current key and move them back to the right. The variable `Top5Bits` would commonly be referred to as a "mask". It has bits set to match just those bits that are required from some other bit pattern; to get the bits you want (from `Result`) you "perform an and operation with the mask".

The bits in the key are moved left five places by the statement:

```
Result = Result << kleftmove;
```

Then the saved leftmost bits are fed back in, followed by the next character from the string:

```
Result ^= Carry;
Result ^= str[i];
```

The following outputs show the encodings obtained for some example strings:

```
mellow :                          0xdc67bd97
yellow :                          0xf467bd97
meadow :                          0xdc611d97
callow :                          0xc027bd97
shallow :                         0x1627bd8b
shade :                           0x70588e5
2,4,6-trinitrotoluene :           0xabe69e14
2,4,5-trinitrotoluene :           0xabe59e14
Bailey, Beazley, and Bradley :    0x64c55ad0
Bailey, Beazley, and Bradney :    0x64c552d0
```

This small sample of test strings doesn't have any cases where two different strings get the same key, but if tried a (much) larger sample of strings you would find some cases where this occurred.

Checking a hash key is quicker than comparing strings character by character. A single comparison of the key values `0x64c55ad0` and `0x64c442d0` reveals that two strings (the two B, B, & Bs) are dissimilar. If you were to use `strcmp()`, the loop would have to compare more than twenty pairs of characters before the dissimilarity was noted.

If you have a table of complex strings that has to be searched many times, then you could gain by generating the hash codes for each string and using these values during the search process. The modified table would have structs that contained the hash key and the string. The table might be sorted on hash key value and searched by binary search, or might simply be searched linearly. A hash key would also be generated for any string that was to be found in the table. The search would check test the hash keys for equality and only perform the more expensive string match in the (few) cases where the hash keys matched.

You can make the lookup mechanism even faster. A hash key is an integer, so it could represent the index of where an item (string) should be stored in an array. The array would be initialized by generating hash keys for each of the standard strings and then copying those strings into the appropriate places in the table. (If two strings have the same hash key, the second one to be encoded gets put into the next empty slot in the array.) When a string has to be found in the table, you generate the hash key and look at that point in the array. If the string there didn't match, you would look at the strings

in the next few locations just in case two strings had ended up with the same key so causing one to be placed at a location after the place where it really should go.

Of course there is a slight catch. A hash key is an integer, but it is in the range 0… 4000million. You can't really have an array with four thousand million strings.

18.2.2 A simple "hash table"

The idea of using the hash key as a lookup index is nice, even though impractical in its most basic form. A slightly modified version works reasonably well.

Modulo arithmetic

The computed hash key is reduced so that instead of ranging from zero to four thousand million, its range is something more reasonable – zero to a few hundred or few thousand. Tables (arrays) with a few thousand entries are quite feasible. The hash key can be reduced to a chosen range 0…N-1 by taking its value modulo N:

```
key = HashString(data);

key = key % N;
```

Hash collisions

Of course that "folds together" many different values. For example, if you took numbers modulo 100, then the values 115, 315, 7915, 28415 etc all map onto the same value (15). When you reduce hash keys modulo some value, you do end up with many more "hash collisions" where different data elements are associated with the same final key. For example, if you take those key values shown previously and reduce them modulo 40, then the words meadow and yellow "collide" because both have keys that are converted to the value 7.

The scheme outlined in the previous subsection works with these reduced size keys. We can have a table of strings, initially all null strings (just the '\0' character in each). Strings can be inserted into this table, or the table can be "searched" to determine whether it already contains a string. (The example in the next section illustrates an application using essentially this structure).

Figure 18.1 illustrates the structure of this slightly simplified hash table. There is an array of one thousand words (up to 19 characters each). Unused entries have '\0'; used entries contain the strings. The figure shows the original hash keys as generated by the function given earlier; these are reduced modulo 1000 to get the index.

The following data structures and functions are needed:

```
const int kLSIZE         = 20;
const int kTBLSIZE       = 1000;

typedef char LongWord[kLSIZE];

LongWord theTable[kTBLSIZE];
```

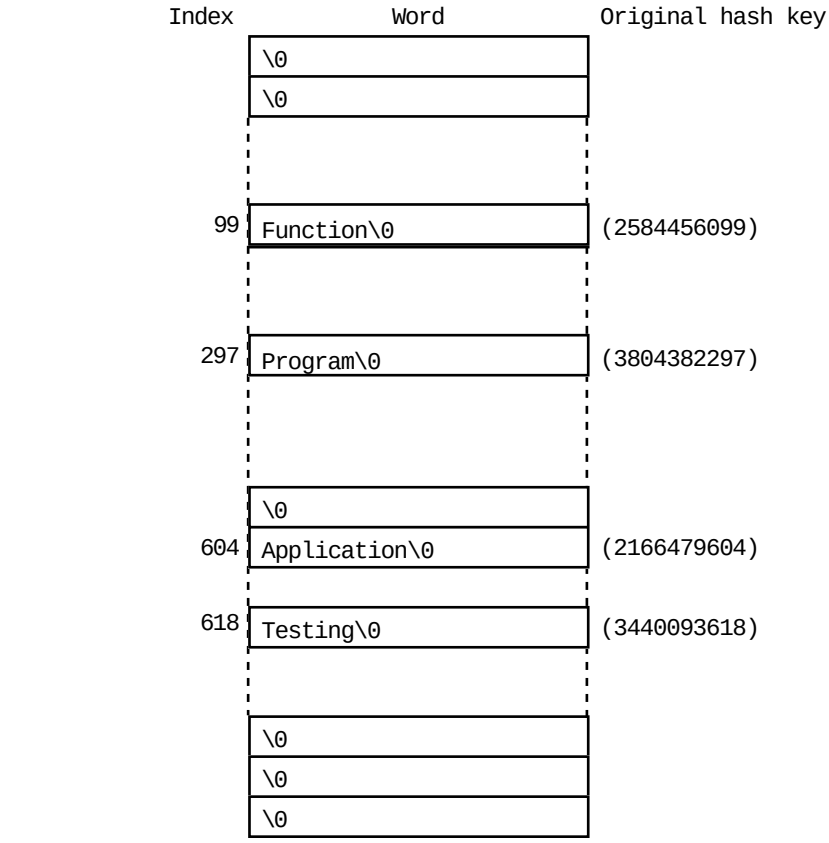

Figure 18.1 A simplified form of hash table.

```
void InitializeHashTable(void);
    Initializes all table entires to '\0'
int NullEntry(int ndx);
    Checks whether the entry at [ndx] is a null string
int MatchEntry(int ndx, const char str[]);
    Checks whether the entry at [ndx] equals str (strcmp())
int SearchForString(const char str[]);
    Searches table to find where string str is located,
    returns position or -1 if string not present
void InsertAt(int ndx, const char str[]);
    Copies string str in table entry [ndx]
int InsertString(const char str[]);
    Organizes insertion, finding place, calling InsertAt();
```

```
    returns position where data inserted, or -1 if table full.
```

The code for these functions is simple. The "initialize table" function sets the leading byte in each word to '\0' marking the entry as unused.

Initializing the table

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++)
           theTable[i][0] ='\0';
}
```

The insertion process has to find an empty slot for a new string, function "null entry" tests whether a specified entry is empty. Function "match entry" checks whether the contents of a non-empty slot matches a sought string.

Checking entries in the table

```
int NullEntry(int ndx)
{
    return (theTable[ndx][0] == '\0');
}

int MatchEntry(int ndx, const char str[])
{
    return (0 == strcmp(str, theTable[ndx]));
}
```

Insertion of entries is handled by the two functions "insert at" and "insert string". Function "insert at" simply copies a string into a (previously empty) table location:

Inserting an entry at its proper position

```
void InsertAt(int ndx, const char str[])
{
    assert (strlen(str) < kLSIZE);
    strcpy(theTable[ndx],str);
}
```

The insert string function does most of the work. First, it "hashes" the string using the function shown earlier to get a hash key. This key is then reduced modulo the length of the array to get a possible index value that defines where the data might go.

The function then has a loop that starts by looking at the chosen location in the array. If this location is "empty" (a null string), then the word can be inserted at that point. The "insert at" function is called, and the insertion position returned.

If the location is not empty, it might already contain the same string (the same word may be being entered more than once). This is checked, and if the word does match the table entry then the function simply returns its position.

In other cases, a "collision" must have occurred. Two words have the same key. So the program has to find an empty location nearby. This function uses the simplest scheme; it looks at the next location, then the one after that and so forth. The position

indicator is incremented each time around the loop; taking its value modulo the array length (so on an array with one thousand elements, if you started searching at 996 and kept finding full, non-matching entries, you would try entries 997, 998, 999, then 0, 1, etc.)

Eventually, this search should either find the matching string (if the word was entered previously), or find an empty slot. Of course, there is a chance that the array is actually completely full. If you ever get back to looking at the same location as where you started then you know that the array is full. In this case the function returns the value -1 to indicate an error.

Finding the place to insert a string

```
int InsertString(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSIZE;
    int pos = k;

    int startpos = pos;

    for(;;) {
           if(NullEntry(pos)) {
                  InsertAt(pos, str);
                  return pos;
                  }
           if(MatchEntry(pos, str)) return pos;
           pos++;
           if(pos >= kTBLSIZE)
                  pos -= kTBLSIZE;
           if(pos == startpos)
                  return -1;
           }
}
```

The function that searches for a string uses a very similar strategy:

"Looking up" a string

```
int SearchForString(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSIZE;
    int pos = k;
    int startpos = pos;
    for(;;) {
           if(NullEntry(pos)) return -1;
           if(MatchEntry(pos, str)) return pos;
           pos++;
           if(pos >= kTBLSIZE)
                  pos -= kTBLSIZE;
           if(pos == startpos)
                  return -1;
           }
```

```
}
```

Although it works, this is not a particularly good implementation of a hash table. The main problem is that it wastes a lot of space with all those "null words". A better implementation is given later after pointers have been introduced.

Another problem with the implementation is the "linear search" strategy used to find an empty slot after a collision of hash keys. This strategy tends to result in clustering of entries and more costly searches. There are alternative strategies that you will be shown in more advanced courses on "data structures and algorithms".

You want to avoid a hash table getting too close to being full. The fuller it is, the greater the chance of hash collisions and then lengthy sequential searches through subsequent table entries. The usual advice is the table should be no more half full when you've loaded all the data. So, you should have a table of about five thousand entries if you need handle a vocabulary of a couple thousand words.

18.2.3 Example: identifying the commonly used words

Specification

Write a program that will produce a table giving in order the fifty words that occur most frequently in a text file, and details of the number of times that each occurred.

The program is to:

1. Prompt for a file name, and then to either open the file or terminate if the file cannot be accessed.

2. Read characters from the file. All letters are to be converted to lower case. Sequences of letters are to be assembled into words (maximum word length < 20 characters). A word is terminated by any non-alphabetic character.

   (Non-alphabetic characters are discarded, they serve only as word terminators.)

3. Save all distinct words and maintain a count with each; this count is initialized to 1 on the first occurrence and is incremented for each subsequent occurrence.

4. When the file has been read completely, the words are to be sorted by frequency and details of the fifty most frequent words are to be printed.

Design

First iteration

The program will have a number of distinct phases. First, it has to read the file identifying the words and recording counts. If we had a hash table containing little structures with words and counts, an insert of a new word could initialize a count while

an insert of an already existing word would update the associated count. A hash table makes it easy to keep track of unique words. These word and count data might have to be reorganised when the end of the file is reached; the structs would need to be moved into an array that can be sorted easily using a standard sort. Next, the data would get sorted (modified version of `Quicksort()` from Chapter 13 maybe?). Finally, the printouts are needed. An initial sketch for `main()` would be:

```
open file
get the words from file
reorganize words
sort
print selection
```

Most of these "functions" are simple. We've dealt with file opening many times before. The hard part will be getting words from the file. While we still have to work out the code for this function, we do know that it is going to end up with a hash table some of whose entries are "null" and others are words with their counts. The sort step is going to need an array containing just the data elements that must be sorted. But that is going to be easy, we will just have to move the data around in the hash table so as to collect all the words at the start of the array. Sorting requires just a modification of Quicksort. The print out is trivial.

```
open file
    prompt for filename
    attempt to open file
    if error
           print warning message and exit

reorganize words
    i = 0
    for j = 0 ; j < hash-table-size; j++
           if(! null_entry(j))
                  entry[i] = entry[j], i++

sort
    modified version of quicksort and partition
    uses an array of word structures,
    sorting on the "count" field of these structures
    (sorts in ascending order so highest frequency in
           last array elements)

print selection
    (need some checks, maybe less than 50 words!)
    for j = numwords-1, j>= numwords - 50, j--
           print details of entry[j]
```

The sort routine will need to be told the number of elements. It would be easy for this to be determined by the "reorganize" routine; it simply has to return the value of its 'i' counter.

These functions shouldn't require any further design iterations. They can be coded directly from these initial specifications. With "sort" and "open file" it is just a matter of "cutting and pasting" from other programs, followed by minor editing. The data type of the arguments to the `Quicksort()` function must be changed, and the code that accesses array elements in `Partition()` must be adjusted. These elements are now going to be structs and the comparison tests in `Partition()` will have to reference specific data members.

The "get the words" process requires more analysis, but the basic structure of the function is going to be something simple like:

```
"get the words"
    Initialize hash table
    while get another word
           insert word
```

(It would be sensible for the insertion to be checked; the program should stop if the hash table has become full causing the insert step to fail.)

Second iteration through design

The program needs slightly modified versions of the hash table functions given in the last section. There is no need for a search function. The other functions used to manipulate the hash table have to be changed to reflect the fact that the table entries aren't just character arrays, instead they are little structs that contain a character array and a count. The main change will be in the insert function. If a word is already in the array, its count is to be updated:

```
insert word
    Get hash key
    reduce key modulo array size
    initialize search position from reduced key
    loop
           if entry at position is null
                  insert at …
                  return position
           if entry at position has string equal to argument
                  update count in entry at position
                  return position
           increment position (mod length of array)
           check for table full (return failure -1 indicator)
```

The "get word" function should fill in a character array with letters read from the file. The function should return true if it finds a word, false if there is no word (this will only occur at the end of file). When called, the function should start by discarding any leading whitespace, digit, or punctuation characters. It can start building up a word with the first letter that it encounters.

Successive letters should be added to the word; before being added they must be converted to lower case. This letter adding loop terminates when a non-letter is found. The function has to check for words in the file that are larger than the maximum allowed for. The program can be terminated if excessive length words are found.

The function has to put a '\0' after the last letter it adds to the word.

These considerations result in the following sketch for "get word":

```
get word (given reference word argument to fill in)
    initialize word[0] to '\0'
                (just in case there is nothing left in file)
    do
           get character
           if at end of file
                  return
    while character isn't a letter

    while character is letter
           add to word
           check word length exceeded
           get next character
           if end of file
                  break loop
    add '\0' to word
```

Third iteration

All that remains is to decide on data structures and function prototypes. We need a struct that combines a character array and a count:

```
const int kLSIZE         = 20;
typedef char LongWord[kLSIZE];
struct WordInfo {
    LongWord      fWord;
    long          fCount;
};
```

Twenty characters (19 + terminating '\0') should suffice for most words. The struct can have a `LongWord` and a long count.

The "hash table" will need a large array of these structs:

```
const int kTBLSIZE       = 4000;
WordInfo gTable[kTBLSIZE];
```

(This `gTable` array needs about 90,000 bytes. Symantec 8 on the PowerPC handles this; but Symantec 7 on a Mac-Quadra cannot handle static data segment arrays this large. You will also have problems in the Borland environment; you will have to use 32-bit memory addressing models for this project.)

The only other global data element would be the `ifstream` for the input file.

Function prototypes

```
Bits HashString(const char str[]);
    Hashing function shown earlier

void InitializeHashTable(void);
    Modified version of function given previously.  Initializes
    both data members of all structs in "hash table" array;
    struct's fWord character array set to null string, fCount
    set to zero.

int NullEntry(int ndx);
int MatchEntry(int ndx, const char str[]);
    Modified versions of functions given previously.  Use fWord
    data member of struct.

void InsertAt(int ndx, const char str[]);
    Modified version of function given previously.  Uses
    strcpy() to set fWord data member of struct; initializes
    fCount data member to 1.

int InsertWord(const char str[]);
    Organize insertion of new word, or update of fCount data
    member for existing word.

void OpenFile();

int GetWord(LongWord& theWord);
    Fills word with next alphabetic string from file
(terminates
    program if word too large).

void GetTheWords(void);
    Organize input or words and insertion into hash table.

int CompressTable(void);
    Closes up gaps in table prior to sorting.

int Partition( WordInfo d[], int left, int right);
    Quicksort's partitioning routine a for a WordInfo array.

void Quicksort( WordInfo d[], int left, int right);
    Modified Quicksort.

void PrintDetails(int n);
    Prints the 50 most frequent words (or all words if fewer
    than 50) with their counts.

int main();
```

Implementation

Only a few of the functions are shown here. The rest are either trivial to encode or are minor adaptations of functions shown earlier.

All the hash table functions from the previous section have modifications similar to that shown here for the case of `InitializeHashTable()`:

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++) {
           gTable[i].fWord[0] ='\0';
           gTable[i].fCount = 0;
    }
}
```

The "insert" function has slightly modified behaviour:

```
int InsertWord(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSIZE;
    int pos = k;

    int startpos = pos;

    for(;;) {
           if(NullEntry(pos)) {
                  InsertAt(pos, str);
                  return pos;
                  }
           if(MatchEntry(pos, str)) {
                  gTable[pos].fCount++;
                  return pos;
                  }
           pos++;
           if(pos >= kTBLSIZE)
                  pos -= kTBLSIZE;
           if(pos == startpos)
                  return -1;
           }
}
```

The `GetWord()` function uses two loops. The first skips non alphabetic characters; the second builds the words:

```
int GetWord(LongWord& theWord)
{
    int n = 0;
    char ch;
```

```
    theWord[0] = '\0';

    do {
            gDataFile.get(ch);
            if(gDataFile.eof())
                    return 0;
    } while (!isalpha(ch));

    while(isalpha(ch)) {
            theWord[n] = tolower(ch);
            n++;
            if(n==(kLSIZE-1)) {
                    cout << "Word is too long" << endl;
                    exit(1);
            }
            gDataFile.get(ch);
            if(gDataFile.eof())
                    break;
    }
    theWord[n] = '\0';
    return 1;
}
```

The `GetTheWords()` function reduces to a simple loop as all the real work is done by the auxiliary functions `GetWord()` and `InsertWord()`:

```
void GetTheWords(void)
{
    InitializeHashTable();
    LongWord aWord;
    while(GetWord(aWord)) {
            int pos = InsertWord(aWord);
            if(pos < 0) {
                            cout << "Oops, table full" << endl;
                            exit(1);
                    }
            }
}
```

The `CompressTable()` function shifts all entries down through the array. Index i identifies the next entry to be filled in. Index j runs up through all entries. If a non null entry is found, it is copied and i incremented:

```
int CompressTable(void)
{
    int i = 0;
    int j = 0;
    for(j = 0; j < kTBLSIZE; j++)
            if(!NullEntry(j)) {
                    gTable[i] = gTable[j];
```

```
                i++;
                }
    return i;
}
```

The `Partition()` function has minor changes to argument list and some statements:

```
int Partition( WordInfo d[], int left, int right)
{
    int val =d[left].fCount;
    int lm = left-1;
    int rm = right+1;
    for(;;) {
           do
                  rm--;
           while (d[rm].fCount > val);

           …
           …
           …
           }
}
```

Function `PrintDetails()` uses a loop that runs backwards from the last used entry tin the table, printing out entries:

```
void PrintDetails(int n)
{
    int min = n - 50;
    min = (min < 0) ? 0 : min;
    for(int i = n-1, j = 1; i >= min ; i --, j++)
           cout << setw(5) << j << ": "
                  << gTable[i].fWord << ", \t"
                         << gTable[i].fCount << endl;
}
```

As usual, `main()` simplifies to a sequence of function calls:

```
int main()
{
    OpenFile();
    GetTheWords();
    int num = CompressTable();
    Quicksort(gTable, 0, num - 1);
    PrintDetails(num);
    return 0;
}
```

Part of the output from a run against a test data file is:

```
Enter filename
txtf.txt
    1: the,     262
    2: of,      126
    3: to,      104
    4: a,       86
    5: and,     74
    6: in,      70
    7: is,      50
    8: that,    41
    9: s,       39
   10: as,      32
   11: from,    30
   12: it,      29
   13: by,      29
   14: are,     27
   15: for,     27
   16: be,      27
```

Most of these are exactly the words that you would expect. What about things like 's'? You get oddities like that. These will be from all the occurrence of 's at the ends of words; the apostrophe ended a word, a new word starts with the s, then it ends with the following space.

Because the common words are all things like "the", they carry no content information about the document processed. Usually programs used to analyze documents have filters to eliminate these standard words. An exercise at the end of this chapter requires such an extension to the example program.

18.3 CODING "PROPERTY VECTORS"

Where might you want actual bit data, that is collections of bits that individually have meaning?

One use is in information retrieval systems.

Suppose you have a large collection of news articles taken from a general science magazine (thousands of articles, lengths varying from 400 to 2000 words). You want to have this collection arranged so that it can be searched for "articles of interest".

Typically, information retrieval systems allow searches in which articles of interest are characterized as those having some minimum number of keywords from a user specified group of keywords. For example, a request might require that at least four of the following keywords be found in an article:

```
aids, hiv, siv, monkey, virus, immune
```

This example query would find articles describing scientific studies on the HIV virus and/or the related virus that infects monkeys (SIV).

You wouldn't want the search program to read each article into memory and check for occurrences of the user specified keywords, that would be much too slow. However, if the keywords used for searches are restricted to those in a predefined set, then it is fairly easy to implement search schemes that are reasonably fast.

These search schemes rely on indexes that relate keywords to articles. The simplest approach is illustrated in Figure 18.2. The main file contains the source text of the news articles. A second "index file" contains small fixed size data records, one for each article in the main file.

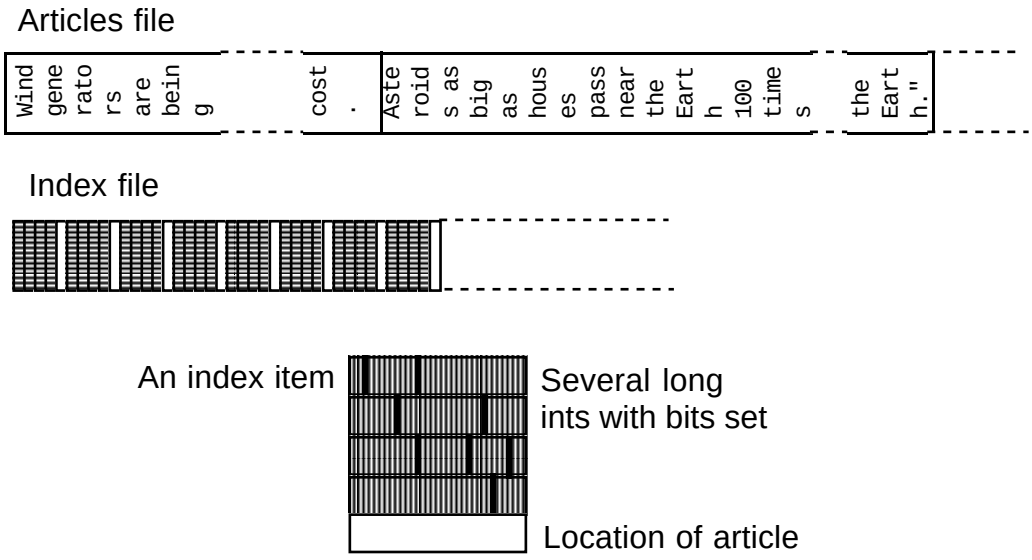

Figure 18.2 Simple Information Retrieval system.

Each index record contains a "bit map". Each bit corresponds to one of the keywords from the standard set. If the bit is '1' it means that the article contained that keyword; if the keyword is not present in the article, the corresponding bit in the map is a zero. Because the articles vary in size, it is not possible to work out where an article starts simply from knowing its relative position in the sequence of articles. So, the index record corresponding to an article contains details of where that article starts in the main file.

Two programs are needed. One adds articles to the main articles file and appends index records to the index file. The second program performs the searches.

Both programs share a table that identifies the (keyword, bit number) details. It is necessary to allow for "synonyms". For example, the articles used when developing

this example included several that discussed chimpanzees (evolution, ecology, social habits, use in studies of AIDS virus, etc); in some articles the animals were referred to as chimpanzees, in others they were "chimps". "Chimp" and chimpanzee are being used as synonyms. If both keywords "chimpanzee" and "chimp" are associated with the same bit number, it is possible to standardise and eliminate differences in style and vocabulary.

A small example of a table of (keyword, bit numbers) is:

```
typedef char LongWord[20];

struct VocabItem {
    LongWord      fWord;
    short         fItemNumber;
};

VocabItem vTable[] = {
    { "aids", 0 },
    { "hiv", 1 },
    { "immunity", 2 },
    { "immune", 2},
    { "drug", 3 },
    { "drugs", 3 },
    { "virus", 4 },
    …
};
```

A real information retrieval system would have thousands of "keywords" that map onto a thousand or more standard concepts each related to a bit in a bit map. The example here will be smaller; the bit maps will have 128 bits thus allowing for 128 different key concepts that can be used in the searches.

Adding data to the file

The program used to add articles to the file would start by using the data in the `VocabItem` table to fill in some variation on the hash table used in the last example. The various files needed would be opened (index file, articles file, text file with new article). Words can then be read from the source text file using code similar to that in the last example (the characters can be appended to the main articles file as they are read from the source text file). The words from the file are looked up in the hash table. If there is no match, the word is discarded. If the word is matched, then the word is one of the keywords; its bit number is taken the matched record and used to set a bit in a bit map record that gets built up. When the entire source text has been processed, the bit map and related location data get appended to the index file.

Running a query

The query program starts by prompting the user to enter the keywords that are to characterize the required articles. The user is restricted to the standard set of keywords as defined in the `VocabItem` table; this is easy to do by employing code similar to the "pick keyword" function that has been illustrated in earlier examples. As keywords are picked, their bit numbers are used to set appropriate bits in a bit map. Once all the keywords have been entered, the user is prompted to specify the minimum number of

matches. Then each index record gets read in turn from the index file. The bit map for the query and that from the index file are compared to find the number of bits that are set in both. If this number of common bits exceeds the minimum specified, then the article should be or interest. The "matched" article is read and displayed.

Bitmaps

The bitmaps needed in this application will be simply small arrays of unsigned long integers. If we have to represent 128 "concepts", we need four long integers. If you go down to the machine code level, you may find that a particular machine architecture defines a numbering for the bits in a word. But for a high level application like this, you can chose your own. The chosen coding is shown in Figure 18.3.

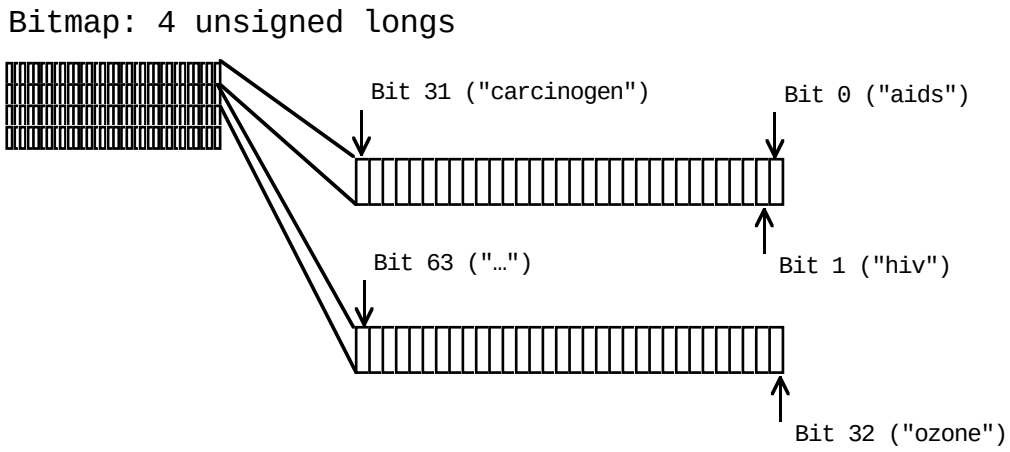

Figure 18.3 Bit maps and chosen bit numbering.

Setting a required bit

In order to set a particular bit, it is necessary to first determine which of the unsigned longs contains that bit (divide by 32) and then determine the bit position (take the bit number modulo 32). Thus bit 77 would be the 13th bit in bitmap[2].

Counting the bits in common

Bits in common to two bit maps can be found by "Anding" the corresponding array entries, as shown in Figure 18.4 The number of bits in common can be calculated by adding up the number of bits set in the result. There are several ways of counting the number of bits set in a bit pattern. The simplest (though not the fastest) is to have a loop that tests each bit in turn, e.g. to find the number of bits in an unsigned long `x`:

```
int count = 0;
int j = 1;
for(int i=0;i<32;i++) {
    if(x & j)
           count++;
    j = j << 1;
    }
```

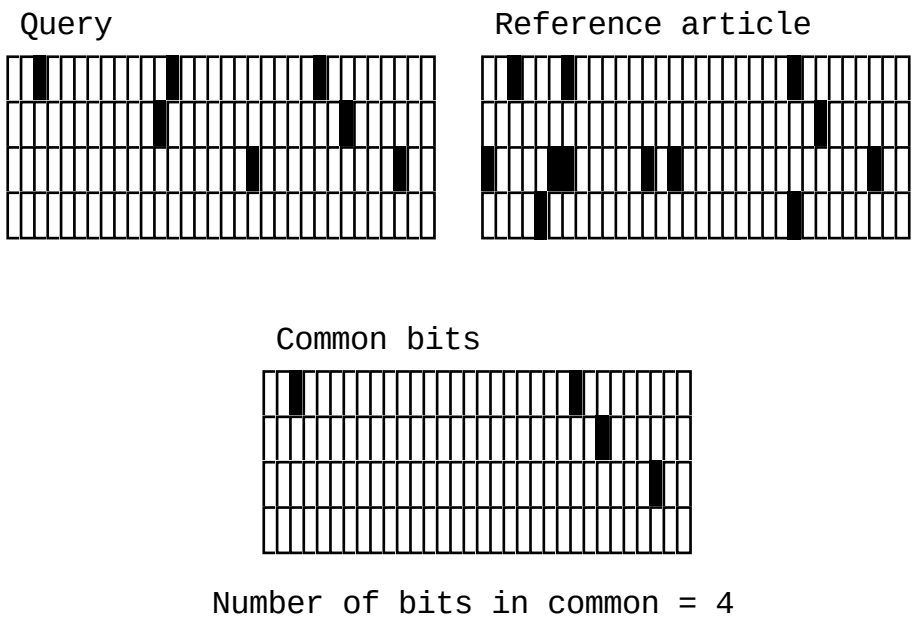

Figure 18.4 Finding and counting the common bits.

Specification

Implement an article entry program and a search program that together provide an information retrieval system that works in the manner described above.

Design

These programs are actually easier than some of the earlier examples! The first sketches for `main()`s for the two programs are:

```
article addition program
    open files
    process text
    close files
```

and

```
search program
    open files(
    get the query
    search for matches
```

Addition program

The open files routine for the addition program has to open an input file with the source text of the article that is to be processed and two output files the main articles file and the index file. New information is to be appended to these files. The files could be opened in "append mode" (but this didn't work with one of the IDEs which, at least in the version used, must have a bug in the iostream run time library). The files can be opened for output specifying that the write position be specified "at the end" of any existing contents.

```
open file
    prompt for and open file with text input
           (terminate on open error)
    open main and index files for output, position "at end"
           (terminate on open error)
```

The close files routine will be trivial, it will simply close all three files.

The main routine is the "process text" function. This has to initialize the hash table, then fill it with the standard words, before looping getting and dealing with words from the text file. When all the words have been dealt with, an assembled bit map must be written to the index file; this bit map has to be zeroed out before the first word gets dealt with.

```
process text file
    initialize and load up hash table
    zero bit map
    while Get Word from file
           deal with word
    write bit map and related info
```

Hash table

The hash table will be like that in the example in the previous section. It will contain small structs. This time instead of being a word and a count, they are "vocab items" that consist of a name and a bit number.

The words are easy to "deal with". The hash table is searched for a match, if one is found its bit number is set in the bit map.

Search program

The open files routine for this program needs to open both index and articles file. No other special actions are required.

The task of "getting the query" can be broken down as follows:

```
get the query
```

```
zero bit map representing query
loop
        get a keyword
        set appropriate bit in bit map
until no more keywords needed
ask for number of keys that must match
```

The loop will be similar to those in previous examples. After each keyword is dealt with, the user will be asked for a Yes/No response to a "More Keywords?" prompt.

The "search for matches" routine will have roughly the following structure:

```
search for matches
    while not end of file on index file
            get the bit map of an index record from file (and
                    details of where article is located)
            get number of bits in common between index record's
                    bit map and query bit map
            if number matched exceeds minimum
                    show matching article
```

The "show matching article" function will be something like:

```
show match
    move "get pointer" to appropriate position in articles'
file
    read character
    while not end marker
            print character
            read next character
```

The articles in the file had better be separated by some recognizable marker character! A null character ('\0') would do. This had better be put at the end of each article by the addition program when it appends data to that file. The "show matching article" function would probably need some extra formatting output to pretty things up. It might be useful if either it, or the calling search routine, printed details of the number of keywords matched.

Bitmaps and bit map related functions

Both programs share a need for bit maps and functions to do things like zero out the bit maps when initializing things, setting chosen bits, and counting common bits. These requirements can be met by having a small separate package that deals with bitmaps. This will have a header file that contains a definition of what a bit map is and a list of function prototypes. A second implementation file will contain the function definitions.

A bit map will be a small array of unsigned longs. This can be specified using a typedef in the header file. The required functions seem to be:

```
Zero bits
    clear all bits in the bit map

Set bit
    work out which array element and which bit
    or a 1 into the appropriate bit

Count common bits
    build a temporary bit map that represents the "And"
           of two bit maps
    for each array element in temporary bit map
           count its bits and add to overall total
    return overall total
```

Further design steps

Many of the functions in both programs are already either simple enough to be coded directly, or are the same as functions used in other examples (e.g. the `YesNo()` function used when asking for more keywords in the search program). A few require further consideration.

The "Get Word" function for the addition program can be almost identical to that in the example in 18.2. The only addition is that every character read must be copied to the output articles file.

The "process text" function needs a couple of additions:

```
process text file
    initialize and load up hash table
    zero bit map
    note the position of the current "end of file"
           of articles file
    while Get Word from file
           deal with word
    write a terminating null to the articles file
    write bit map and related info (i.e. the position
           of previous end of file!)
```

These additions make certain that there is a null character separating articles as required by the search program's "show match" function. The other additions clarify the "related info" comment in the original outline. Each record written to the index file has to contain the location of the start of the corresponding article as well as a bit map. So the "process" function had better note the current length of the articles file before it adds any words. This is the "related info" that must then be written to the index file.

The "get a keyword" function needed in search can be modelled on the `PickKeyWord()` function and its support routines developed in Section 12.4. Apart from `PickKeyWord()` itself, there were the associated routines `FindExactMatch()`,

`CountPartialMatches()` and `PrintPartialMatches()`. All of these routines have to be reworked so that they use an array of `VocabItems` rather than a simple array of strings. This recoding is largely a matter of changing the data type of arguments and adding a data member name to some references to array elements.

Final steps in design

Here this is simply a matter of identifying global (and filescope data), deciding on how to deal with the VocabItem array needed in both programs, and finalising the function prototypes.

Both programs share the array of `VocabItems` (the structs with keywords and bit numbers), and they also both need a count of the number of `VocabItems` defined. This information should be in a separate file that can be #included by both programs:

Filevocab.inc

```
/*
 Vocabulary file for information retrieval example.
*/

typedef char LongWord[20];

struct VocabItem {
    LongWord      fWord;
    short  fItemNumber;
};

VocabItem vTable[] = {
    { "aids", 0 },
    { "hiv", 1 },
    …
    …
    { "cancer", 28 },
    { "tumour", 29 },
    { "therapy", 30 },
    { "carcinogen", 31 },
    { "ozone", 32 },
    { "environment", 33 },
    { "environmental", 33 },
    …
    …
    { "toxin", 50 },
    { "poison", 50 },
    { "poisonous", 50 },
    …
};

int NumItems = sizeof(vTable) / sizeof(VocabItem);
```

The `vTable` array and the count `NumItems` will be "globals" in both the programs.

Both programs will require a number of `ifstream` and/or `ofstream` variables that get attached to files. These can be globals.

The addition program requires a "hash table" whose entries are `VocabItems`. This won't be particularly large as it only has to hold a quick lookup version of the limited information in the `vTable` array.

The search program could use globals for the bit map that represents a query and for the minimum acceptable match.

The typedef defining a "bit map" would go in a header file along with the associated function prototypes:

File mybits.h

```
#ifndef __MYBITS__
#define __MYBITS__

#define MAXBIT    127
#define MAPSZ     4

typedef unsigned long Bits;

typedef Bits Bitmap[MAPSZ];


void ZeroBits(Bitmap& b);

void SetBit(int theBit, Bitmap& b);

int CountCommonBits( Bitmap& b1,  Bitmap& b2);

#endif
```

The function prototypes for the remaining functions in the two programs are:

article addition program

```
Bits HashString(const char str[]);
    The hash function given earlier.

void InitializeHashTable(void);
    Fills hash table with "null VocabItems" (fWord field
    == '/0', fItemNumber field == -1).

int NullEntry(int ndx);
int MatchEntry(int ndx, const LongWord w);
void InsertAt(int ndx, const VocabItem v);
    Similar to previous examples except for use of VocabItem
    structs.

int SearchForWord(const LongWord w);
    Minor variation on previously illustrated hash table
    search function.

int InsertVocabItem(const VocabItem v);
```

```
        Inserts standard VocabItem into hash table.

    void InsertKeyWords(void);
        Loops through all entries in vTable, inserting copies into
        hash table.

    void OpenFiles(void);
    void CloseFiles(void);

    int GetWord(LongWord& theWord);
        Similar to previous get word, just copies input characters
    to output file in addition to other processing.

    void ProcessText(void);
        Main loop of addition program.

    int main();
```

search program

```
    void OpenFiles(void);

    int FindExactMatch(const VocabItem keyws[], int nkeys,
        const LongWord input);
    int CountPartialMatches(const VocabItem keyws[], int nkeys,
        const LongWord input, int& lastmatch);
    void PrintPartialMatches(const VocabItem keyws[], int nkeys,
        const LongWord input);
    int PickKeyWord(const VocabItem keywords[], int nkeys);
        These functions are minor variations of those defined
        in 12.4

    int YesNo(void);

    void GetTheQuery(void);
        Loop building up bit map that represents the query.

    void ShowMatch(long where);
        Prints article starting at byte offset 'where' in main
        articles file.

    void SearchForMatches(void);

    int main();
```

Implementation

Only a few of the functions are shown here. The others are either identical to, or only minor variations, of functions used in earlier examples.

Examples of functions from the mybits.cp file are:

```
void SetBit(int theBit, Bitmap& b)
{
    assert((theBit >= 0) && (theBit <= MAXBIT));
    int word = theBit / 32;
    int pos = theBit % 32;
    int mask = 1 << pos;
    b[word] |= mask;
}
```

Function `SetBit()` uses the scheme described earlier to identify the array element and bit position. A "mask" with one bit set is then built by shifting a '1' into the correct position. This mask is then Or-ed into the array element that must be changed.

Function `CountCommonBits()` Ands successive elements of the two bit patterns and passes the result to function `CountBits()`. The algorithm used by `CountBits()` was illustrated earlier.

```
int CountCommonBits( Bitmap& b1,  Bitmap& b2)
{
    int result = 0;
    for(int i = 0; i < MAPSZ; i++) {
           Bits temp = b1[i]  & b2[i];
           result += CountBits(temp);
           }
    return result;
}
```

The standard hash table functions all have minor modifications to cater for the different form of a table entry:

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++) {
           gTable[i].fWord[0] ='\0';
           gTable[i].fItemNumber = -1;
    }
}
```

The `OpenFiles()` function for the articles addition program has a mode that is slightly different from previous examples; the `ios::ate` parameter is set so that new data are added "at the end" of the existing data:

```
void OpenFiles(void)
{
    char   fname[100];
    cout << "Enter name of file with additional news article"
                  << endl;
```

```
    cin >> fname;
    gTextFile.open(fname, ios::in | ios::nocreate);
    if(!gTextFile.good()) {
           cout << "Sorry, couldn't open input text file"
                  << endl;
                  exit(1);
           }
    gInfoData.open(InfoFName1, ios::out | ios::ate);
    if(!gInfoData.good()) {
           cout << "Sorry, couldn't open main info. file" <<
                  endl;
                  exit(1);
           }
    gInfoIndex.open(InfoFName2, ios::out | ios::ate);
    …
}
```

Function `GetWord()` is similar to the previous version, apart from the extra code to copy characters to the output file:

```
int GetWord(LongWord& theWord)
{
    int n = 0;
    char ch;
    theWord[0] = '\0';

    do {
           gTextFile.get(ch);
           if(gTextFile.eof())
                  return 0;
           gInfoData.write(&ch,1);
    } while (!isalpha(ch));

    while(isalpha(ch)) {
           …
           …
           gInfoData.write(&ch,1);
    }
    theWord[n] = '\0';
    return 1;
}
```

Most of the work is done by `ProcessText()`:

```
void ProcessText(void)
{
    InitializeHashTable();
    InsertKeyWords();

    LongWord aWord;
```

```
    Bitmap aMap;
    ZeroBits(aMap);
    long   where;
    where = gInfoData.tellp();
    while(GetWord(aWord)) {
           int pos = SearchForWord(aWord);
           if(pos >= 0) {
                  int keynum = gTable[pos].fItemNumber;
                  SetBit(keynum, aMap);
                  }
           }
    char ch = '\0';
    gInfoData.write(&ch, 1);
    gInfoIndex.write(&aMap,sizeof(aMap));
    gInfoIndex.write(&where,sizeof(long));
}
```

The call to `tellp()` gets the position of the end of the file because the open call specified a move to the end. The value returned from `tellp()` is the starting byte address for the article that is about to be added.

The principal functions from the search program are:

```
void GetTheQuery(void)
{
    ZeroBits(gQuery);
    cout << "Enter the terms that make up the query" << endl;
    do {
           int k = PickKeyWord(vTable, NumItems);
           int bitnum = vTable[k].fItemNumber;
           SetBit(bitnum, gQuery);
    }
    while (YesNo());
    cout << "How many terms must match ? ";
    cin >> gMinMatch;

}
```

Function `GetTheQuery()` builds up the query bit map in the global `gQuery` then set `gMinMatch`.

Function `ShowMatch()` basically copies characters from the articles' file to the output. There is one catch here. The file will contain sequences of characters separated by an end of line marker. The actual marker character used will be chosen by the text editor used to enter the original text. This "end of line" character may not result in a new line when it is sent to the output (instead an entire article may get "overprinted" on a single line). This is catered for in the code for `ShowMatch()`. A check is made for the character commonly used by editors to mark an end of line (character with hex representation `0x0d`). Where this occurs a newline is obtained by `cout << endl`. (You

might have to change that hex constant if in your environment the editors use a different character to mark "end of line".)

```
void ShowMatch(long where)
{
    char ch;
    int linepos = 0;
    gDatafile.seekg(where,ios::beg);
    gDatafile.get(ch);
    while(ch != '\0') {
           if(ch == 0x0d)
                  cout << endl;
           else cout << ch;
           gDatafile.get(ch);
    }
    cout << "\n------\n";
}
```

The `SearchForMatches()` function simply reads and checks each record from the index file, using `ShowMatch()` to print any matches. An error message is printed if nothing useful could be found.

```
void SearchForMatches(void)
{
    Bitmap b;
    long wh;
    int matches = 0;
    gIndexfile.seekg(0,ios::beg);
    while(!gIndexfile.eof()) {
           gIndexfile.read(&b, sizeof(Bitmap));
           gIndexfile.read(&wh, sizeof(long));
           int n = CountCommonBits(b, gQuery);
           if(n>=gMinMatch) {
                        cout << "Matched on " << n << " keys"
                               << endl;
                        ShowMatch(wh);
                        matches++;
                 }
    }
    if(matches == 0)
           cout << "No matches" << endl;
}
```

A test file was built using approximately seventy articles from a popular science magazine as input. The results of a typical search are:

```
Enter the terms that make up the query
fusion
Another search term? (Y or N) : y
power
```

```
Another search term? (Y or N) : n
How many terms must match ? 2
Matched on 2 keys
Cold fusion is alive and well and thriving on Japanese money in an
"attractive" part of France, says Martin Fleischmann, the chemistry
professor who in 1989 claimed to have produced nuclear fusion in a
test tube at room temperature.
 Cold fusion said Fleischmann and his American colleague Stanley
…
…
negligible compared with the heat liberated.

------
```

18.4 PIECES (BIT FIELDS)

This is definitely a "read only" topic in C++, and outside of a few text books and some special purpose low-level code it is topic where you won't find much to read. Any low-level code using the features described in this section will relate to direct manipulation of particular groups of bits in the control registers of hardware devices. We will not cover such machine specific detail and consider only the (relatively rare) usage in higher level data structures.

Suppose you have some kind of data object that has many properties each one of which can take a small number of values (mostly the same sorts of thing that you would consider suitable for using enumerated types) e.g.:

```
colour:  coral, pearl, smoke-grey, steel-blue, beige;
size:    small, medium, large, x-large, xx-large,
finish:  matte, silk, satin, gloss
style:   number in range 0..37
```

If you did chose to work with enumerated types, the compiler would make each either a single unsigned character, or an unsigned short integer. Your records would need at least three bytes for `colour`, `size`, and `finish`; another unsigned byte would be needed for the `style`. Using enumerated types, the typical struct representing these data would be at least four bytes, 32-bits, in size.

But that isn't the minimum storage needed. There are five colors, for that you need at most three bits. Another three bits could hold the size. The four finishes would fit in two bits. Six bits would suffice for the style. In principle you could pack those data into 14 bits or two bytes.

This is permitted using "bit fields":

```
#include <stdlib.h>
#include <iostream.h>

struct meany {
```

```
    unsigned colour : 3;
    unsigned size : 3;
    unsigned finish : 2;
    unsigned style : 6;
};

int main()
{
    meany m;

    m.colour = 4;
    m.size = 1;
    m.finish = 2;
    m.style = 15;
    cout << m.style << endl;

    return EXIT_SUCCESS;
}
```

(With the compiler I used, a "meany" is 4 bytes, so I didn't get to save any space anyway.)

Note, you are trading space and speed. For a small reduction in space, you are picking up quite an overhead in the code needed to get at these data fields.

A real advantage of bit fields

There is one possible benefit. Normally, if you were trying to pack several small data fields into a long integer, you would need masking and shift operations to pick out the appropriate bits. Such operations obscure your code.

If you use bit fields, the compiler generates those same masking and shift operations to get at the various bit fields. The same work gets done. But it doesn't show up in the source level code which is consequently slightly easier to read.