

```

#include <ctype.h>
#include "index.h"

#define IsHeader(c)          ((c) == '-')
#define IsString(c)          ((c) == '"')
#define IsEOF(c)             ((c) == EOF)

static int NextToken ();
static int SkipToken ();
static char *ScanString ();
static char *ScanToken ();
static long ScanLong ();
static int CompareCanonical ();

INDEX *CreateIndex ()
{
    INDEX *index;

    if ((index = (INDEX *) malloc (sizeof (INDEX))) == NULL)
        AllocError ("CreateIndex");
    index->n_entries = 0;
    index->n_principals = 0;
    index->root = NULL;

    return (index);
}

INDEX *ReadIndex (index, file)
    INDEX *index;
    register FILE *file;
{
    char buffer[512];
    char *identifier, *file_name;
    long offset, length;
    register int c;

    identifier = file_name = NULL;
    offset = 0;
    length = -1;

    /* skip header */
    if (!IsHeader (NextToken (file)))
        IOError ("ReadIndex - bad master index");
    while (!IsHeader (SkipToken (file)));
    SkipToken (file);

    /* scan successive records for this index (until next header) */
    while (IsString (NextToken (file)))
    {
        identifier = ScanString (file);
        c = NextToken (file);
        if (IsString (c) || IsHeader (c) || IsEOF (c))
            /* if no new file_name, use pointer to last one created */
            InsertInIndex (index, CreateEntry (synonym, identifier, file_name,
offset, length));
        else
            /* next is not an ident or index header, must be file_name */
            {

```

```

        file_name = ScanToken (file);
        c = NextToken (file);
        offset = isdigit (c) ? ScanLong (file) : 0L;
        c = NextToken (file);
        length = isdigit (c) ? ScanLong (file) : -1L;
        InsertInIndex (index, CreateEntry (principal, identifier, file_name,
offset, length));
    }
}

return (index);
}

```

```

/*
Scan up to beginning of next token (i.e., run of non-blank
characters) in file.
*/

```

```

static int NextToken (file)
    register FILE *file;
{
    register int c;

    while (c = getc (file), isspace (c));
    ungetc (c, file);

    return (c);
}

```

```

/*
Scan off next token (run of non-blanks), if any, then scan
up to beginning of the following token.
*/

```

```

static int SkipToken (file)
    register FILE *file;
{
    register int c;

    while (c = getc (file), !isspace (c) && c != EOF);
    while (c = getc (file), isspace (c));
    ungetc (c, file);

    return (c);
}

```

```

/*
Scan off a sequence of characters delimited by double quotes,
placing those characters in a dynamically allocated buffer to be
returned.
*/

```

```

static char *ScanString (file)
    register FILE *file;
{
    register char *bp, *string;
    register int c;
    char buffer[512];

```

```

/* treat "" as a " embedded in the string; just scan through trailing quote */
bp = buffer;
getc (file);
NextToken (file); /* skip any leading blanks */
while (((c = getc (file)) != EOF) &&
        ((c != '"') || ((c = getc (file)) == '"'))
        *bp++ = c;
ungetc (c, file);
while ((bp > buffer) && isspace (*--bp)); /* strip any trailing blanks */
*++bp = '\0';
if ((string = malloc (bp - buffer + 1)) == NULL)
    AllocError ("ScanString");
strcpy (string, buffer);

return (string);
}

```

```

/*
Scan off next token, placing it in a dynamically allocated buffer.
*/

```

```

static char *ScanToken (file)
    register FILE *file;
{
    register char *bp, *string;
    register int c;
    char buffer[512];

    bp = buffer;
    while (c = getc (file), !isspace (c) && c != EOF)
        *bp++ = c;
    ungetc (c, file);
    *bp = '\0';
    if ((string = malloc (bp - buffer + 1)) == NULL)
        AllocError ("ScanToken");
    strcpy (string, buffer);

    return (string);
}

```

```

/*
Scan off next sequence of numerics, returning result as a long integer.
*/

```

```

static long ScanLong (file)
    register FILE *file;
{
    register long n;
    register int c;

    n = 0;
    while (c = getc (file), isdigit (c))
        n = 10 * n + (c - '0');
    ungetc (c, file);

    return (n);
}

```

```

INDEX *InsertInIndex (index, entree)
    INDEX *index;
    ENTRY *entree;
    {
        InsertInTree (&(index->root), entree, CompareCanonical);
        index->n_entries++;
        if (TypeOfEntry (entree) == principal)
            index->n_principals++;

        return (index);
    }

ENTRY *FindInIndex (index, key)
    INDEX *index;
    char *key;
    {
        TREE *tree;

        if ((tree = FindInTree (index->root, key, CompareCanonical)) != NULL)
            return (EntryOfTree (tree));
        else
            return (NULL);
    }

/*
Compare two character strings, ignoring case, leading and trailing blanks,
and runs of embedded blanks, returning negative, 0, or positive for
s1 < s2, s1 == s2, or s1 > s2, respectively.
*/

static int CompareCanonical (s1, s2)
    char *s1, *s2;
    {
        register int c1, c2;

        /* remove leading whitespace */
        while (c1 = *s1, isspace (c1))
            s1++;
        while (c2 = *s2, isspace (c2))
            s2++;

        do
            {
                /* canonicalize next char of s1 */
                c1 = *s1++;
                if (isspace (c1))          /* if whitespace, compress multiple whitespace */
                    {
                        while (c1 = *s1, isspace (c1))
                            s1++;
                        if (c1 != '\0')    /* ignore trailing whitespace */
                            c1 = ' ';
                    }
                else if (isupper (c1))    /* if uppercase, make lowercase */
                    c1 = tolower (c1);
                /* same for s2 */
                c2 = *s2++;
                if (isspace (c2))

```

```

    {
        while (c2 = *s2, isspace (c2))
            s2++;
        if (c2 != '\0')
            c2 = ' ';
    }
    else if (isupper (c2))
        c2 = tolower (c2);
    }
    while (c1 == c2 && c1 != '\0');

return (c1 - c2);
}

```

```

ENTRY *NthInIndex (index, n)
INDEX *index;
long n;
{
    TREE *tree;

    if ((tree = NthInTree (index->root, n)) != NULL)
        return (EntryOfTree (tree));
    else
        return (NULL);
}

```