```
#define MAXCHARS 1000    /* max chars per word */
#define INITIAL   100    /* initial size of word array */

typedef char word_t[MAXCHARS+1];
int getword(word_t W, int limit);
void exit_if_null(void *ptr, char *msg);

int
main(int argc, char *argv[]) {
    word_t one_word;
    char **all_words;
    size_t current_size=INITIAL;
    int numdistinct=0, totwords=0, i, found;
    all_words = (char**)malloc(INITIAL*sizeof(*all_words));
    exit_if_null(all_words, "initial allocation");
    while (getword(one_word, MAXCHARS) != EOF) {
        totwords = totwords+1;
        /* linear search in array of previous words... */
        found = 0;
        for (i=0; i<numdistinct && !found; i++) {
            found = (strcmp(one_word, all_words[i]) == 0);
        }
        if (!found) {
            /* a new word exists, but is there space? */
            if (numdistinct == current_size) {
                current_size *= 2;
                all_words = realloc(all_words,
                    current_size*sizeof(*all_words));
                exit_if_null(all_words, "reallocation");
            }
            /* ok, there is definitely space in array */
            all_words[numdistinct] =
                (char*)malloc(1+strlen(one_word));
            exit_if_null(all_words[numdistinct],
                "string malloc");
            /* and there is also a space for the new word */
            strcpy(all_words[numdistinct], one_word);
            numdistinct += 1;
        }
    }
    printf("%d words read\n", totwords);
    for (i=0; i<numdistinct; i++) {
        printf("word #%d is \"%s\"\n", i, all_words[i]);
        free(all_words[i]);
        all_words[i] = NULL;
    }
    free(all_words);
    all_words = NULL;
    return 0;
}
```

Figure 10.3: Using `realloc` so that an array can grow as large as is required. Function `getword` is defined in Figure 7.13 on page 121.

the first one in an array of pointers to characters. Hence the declared type: `char**`, the same as the `(char*)[]` that is used to describe the program argument `argv`.

Figure 10.3 also shows a second common operation in C – that of using `malloc` to obtain exactly enough space for some particular string to be stored, using `strlen` to determine the length of it. The "`1+`" in that call is to allow for the null byte at the end of the string. Apart from the null, there is no waste space at all in the strings being stored. In this framework, the memory waste caused by over-sizing `all_words` by as much as a factor of two might be more than compensated for by not wasting any space in the stored strings.

When using `malloc` to create an array to store a string, one extra character must be requested, to hold the terminating null byte.

A third point is illustrated by Figure 10.3: the use of a function `exit_if_null` to test each pointer after any of the memory allocation routines has been used. If the allocation fails, the pointer is `NULL`, and program execution should be aborted. The second argument passed to `exit_if_null` is a message to be printed prior to program exit. The flexibility associated with `void*` pointers means that a possible implementation of the function is thus:

```
void
exit_if_null(void *ptr, char *msg) {
    if (!ptr) {
        printf("unexpected null pointer: %s\n", msg);
        exit(EXIT_FAILURE);
    }
}
```

The minimalist guard is possible because `NULL` is equivalent to integer zero, meaning that `!ptr` is one (true) exactly whenever the pointer `ptr` is `NULL`.

A more general way of achieving a similar result is to use the `assert` function specified in the header file `assert.h`:

```
    assert(all_words[numdistinct] != NULL);
```

If the argument expression is false, program execution is halted and a diagnostic message printed indicating the line number and the assertion that has been violated. It is also perfectly reasonable to write:

```
    assert(0 <= numdistinct && numdistinct < current_size);
```

This one checks that two variables are maintaining an expected relationship, and could be used to guard an array access.

The `assert` function is used in the remainder of this book to indicate a property that should always be true at that point in a program, and that if violated, represents a serious problem. You are encouraged to use `assert` in the same way in your programming. On the other hand, a less abrupt handling of `NULL` pointers is sometimes required, in which case a dedicated test should be written. Whichever route is used, the value returned from `malloc` should always be tested before the pointer is used.