



## School of Physics

# A complete example: a simple note-book program

Comments and questions to [John Rowe](#).

So far our examples have been more or less as small as we can get away with to illustrate a particular point. Here we present a small but usable program to show a slightly wider context.

Whenever we program it is important to work one thing at a time. A popular way to do this is to build the program up in stages, so that we always have a working program, even if at the early stages it is too simple to be useful. In fact we will always end up working that way anyway as after we have "finished" a program, if it's any good people will ask us to add new features.

When adding new features to a program we will obviously add brand new functions, but we will typically also add new functionality to existing functions. This in turn mean adding new arguments. There are a number of ways we could approach this:

1. Think ahead and have the arguments from the start.  
Sometimes easier said than done!
2. Copy the function and have two different functions, with different names.  
This is best either when the new functionality is a very large addition or when we know that the function will have two totally different new functionalities added to it.
3. Add a new argument to the function.  
This requires us to go through the entire program adding the new argument to every call and is best for a function that we won't often need to use in its original form.
4. Rename the function, add a new argument and provide a "wrapper" function with the original name and arguments.

We shall demonstrate all four options later on in the lecture.

## How are we going to store the data?

When ever we write a program the first question should always be: "what are the data, what properties do the data have, are there any relationships between them?".

As mentioned last week, since we are storing several character strings that implies a two-dimensional character array. Any individual character string can be accessed as `entries[j]` (one index) and any single character as `entries[j][k]` (two indices). This is so important we will repeat it:

A single index, `entries[j]`, refers to the whole one-dimensional character array `j`, two indices `entries[j][k]` refers to one particular character.

AS the list of notes is going to be used by just about every function in the program it makes sense to have it as a **global array** declared outside and before any of the functions.

As always, trying to store a string that is longer than the character array we are trying to store it in would be disastrous so we shall be careful to use `fgets()` with the appropriate limit on the number of characters.

Finally, we have the ability to store several strings but we also need some way of keeping track of which of our entries is in use (contains a data string) and which isn't. We need this information to tell us where to store a new string and, when printing out our data, which strings to print and which to ignore.

Fortunately there is a very simple solution to this problem. Since all text strings end with a binary zero, `'\0'`, and we do not need to store empty strings, it follows that we can adopt the following convention:

If `entries[j][0]` has the value `'\0'`, that means that (one-dimensional) character array `j` is not in use.

As an extra bonus, since `entries` is an external array C will initialize it all to zeros for us anyway.

This leads to the following code at the start of our program, before any of the functions:

```
/*
 * The entries. If entries[i][0] == '\0' that means
 * that entry[i] is unused.
 */
#define MAXLEN 128
#define MAXENTRIES 256
char entries[MAXENTRIES][MAXLEN]; /* All set to '\0' as it is external */
```

Although this seems simple, it does something very important for us: it declares the storage for our data and gives a simple rule to tell us which of our `MAXENTRIES` strings are in use and which is free.

## Iteration 1: Our first working program

As always, it's best to write a simple, working program with the bare minimum of features and then to add more later.

Here is our first working program, it can read in and store new data in memory but cannot yet delete entries, search through

Technically, "data" is the plural of "datum".

them or store them to disk. We will first show the entire code, then discuss the individual functions one at a time.

```

/*
 * Simple notebook program, one entry per line.
 * For example, each line could contain a name and phone number.
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/*
 * The entries. If entries[i][0] == '\0' that means that entry is
 * unused.
 */
#define MAXLEN 128
#define MAXENTRIES 256
char entries[MAXENTRIES][MAXLEN]; /* All set to '\0' as it is external */

int readoneline(char line[], int maxbytes);
void newentry(void);
void listentries(FILE *);

int main() {
    int choice;

    /* Event loop */
    while ( 1 == 1 ) {

        printf("\nMenu:\n\n"
               "\t1. Add new entry\n"
               "\t2. List all entries\n"
               "\t0. Quit\n");
        scanf("%d", &choice);

        switch (choice) {
        case 0:
            return 0;

        case 1:
            newentry();
            break;

        case 2:
            listentries(stdout);
            break;

        default:
            printf("\nUnknown option \"%d\"\n", choice);
        }
    }
}

/*
 * List all entries
 */
void listentries(FILE *out) {
    int i;

    if (out == NULL) {
        fprintf(stderr, "Cannot write to file\n");
        return;
    }

    for (i = 0; i < MAXENTRIES; ++i)
        if ( entries[i][0] != '\0' )
            fprintf(out, "%s\n", entries[i]);
}

/*
 * Read in and add a new entry
 */
void newentry() {
    int i;

```

```

for (i = 0; i < MAXENTRIES; ++i) /* Find an empty slot */
  if (entries[i][0] == '\0') {
    printf("Please type in the new entry\n");
    readoneline(entries[i], MAXLEN) ;
    return;
  }

/* If we got here the list was full */
printf("Sorry, there is no space for any new entries\n");
}

/*
 * Read in a line of text, looping until it has some non-spaces.
 * Trim the final '\n' the end of the line.
 * Return 1 for success, zero for failure.
 */
int readoneline(char line[], int maxbytes) {
  while ( 1 == 1 ) {
    int i;

    if ( fgets(line, maxbytes, stdin) == NULL )
      return 0; /* Out of data */

    /* Trim the final new-line character */
    i = strlen(line) - 1;
    if ( line[i] == '\n')
      line[i] = '\0';

    /* Look for a non-blank character. */
    for (i = 0; line[i] != '\0'; ++i)
      if ( isspace(line[i]) == 0)
        return 1;
  }
}

```

This has four functions, `main()`, `listentries()` which prints all the entries to the screen, `newentry()`, which reads an entry from the keyboard (calling `readoneline()`), and `readoneline()`, which we have already met.

## main()

The `main()` function is very simple, it just consists of an infinite loop, reading in the user command (Quit, Add a new entry of List existing entries), carrying it out and waiting for the next command. This basic structure is quite a common one and is called the **event loop**.

## listentries()

This is the simplest function, consisting of the loop:

```

for (i = 0; i < MAXENTRIES; ++i)
  if ( entries[i][0] != '\0' )
    fprintf(out, "%s\n", entries[i]);

```

This loops over each of the `MAXENTRIES` one-dimensional character arrays and if `entries[i][0] != '\0'` it prints it out. (Remember that `entries[i][0] == '\0'` means that the  $i^{\text{th}}$  data string is unused.

## Thinking ahead

You will notice that we have allowed `listentries()` to accept a `FILE *` argument and we are calling from `main()` with it with `stdout`. Whilst this may seem unnecessary at this point, it's worth thinking ahead a little and realising that the program will need to be able to save its data to a file. So we put it in now.

However, we have dodged one slightly subtle problem: what happens if the `FILE *` argument, `out`, is `NULL`? Clearly this shouldn't happen and we will fix this problem in a later iteration of the code.

This is option 1 in the list above.

## newentry()

Once we have understood `listentries()`, `newentry()` is quite simple. It also loops over each of the `MAXENTRIES` one-dimensional character arrays but this time, when it finds an empty one it calls `readoneline()` to read a line of text from the screen into that, previously-empty, one-dimensional character array and returns. The main loop is:

```

for (i = 0; i < MAXENTRIES; ++i) /* Find an empty slot */
  if (entries[i][0] == '\0') {
    printf("Please type in the new entry\n");

```

```

    readoneline(entries[i], MAXLEN) ;
    return;
}

```

If it cannot find an empty space for the data it prints a polite message saying it is full.

This program works: we could type in somebody's name and telephone numbers on a single line, the program will remember it and we can repeat the process to store up to MAXENTRIES names and telephone numbers, or short reminders ("buy some milk").

But it has (at least) three obvious problems it's not possible to search, we cannot delete an entry and it does not store the data to disk. We also have the question of what to do if `listentries()` is called with a NULL FILE pointer. We will deal with these one at a time.

## Iteration 2: adding search and delete

### Search

The `listentries()` currently lists all the non-blank entries. Whilst that's sometimes useful, we will often want the ability to search for a particular substring.

We will choose to write a new function, `searchentries()`, although we could equally have chosen modify `listentries()`.

The main reason for choosing to do it this way is that `listentries()` has the ability to write to either the screen or a file whereas `searchentries()` always writes to the screen, and `searchentries()` will have the option to delete entries whereas `listentries()` won't. These tasks are sufficiently different to have two separate functions rather than one larger one.

We can use the `strstr()` function to find if one string contains another.

### searchentries()

In the loop inside `listentries()` we had the test:

```

if ( entries[i][0] != '\0' )
    fprintf(out, "%s\n", entries[i]);

```

to print out all non-blank entries. In `searchentries()` we will now require a second condition to also be true, namely that the search string must be contained in that line, i.e. that `strstr()` must return a non-zero result.

The inner test is now:

```

if ( entries[i][0] != '\0' && strstr(entries[i], searchstr) ) {
    fprintf(stdout, "%s\n", entries[i]);
}

```

The entire function is:

```

/*
 * List entries containing search string,
 */
void searchentries(void) {
    int i;
    char searchstr[MAXLEN];

    printf("Please enter the string to search for (case-sensitive)\n");
    if ( readoneline(searchstr, MAXLEN) == 0 )
        return;

    for ( i = 0; i < MAXENTRIES; ++i )
        if ( entries[i][0] != '\0' && strstr(entries[i], searchstr) ) {
            printf("%s\n", entries[i]);
        }
}

```

The `main()` function has another option added to the menu, since this is straightforward we won't bother to show it.

## Adding a delete function

No notebook is complete without the ability to delete an entry. The basic elements of a delete function are:

1. The ability to choose which entry or entries to delete.
2. A way to delete those entries.

This is option 3 in the list above.

Reminder: the function `strstr(str1, str2)` returns non-zero if the string `str1` does contain `str2`, zero if it does not.

To deal with the latter first, to delete entry `i` all we have to do is to set `entries[i][0]` to zero, as that is our definition of an unused entry.

The first issue is equally easily dealt with: we already have a function, `searchentries()`, that searches for entries that match the one(s) we want. All we have to do is to add the statement:

```
entries[i][0] = '\0' ;
```

As before, we can either choose to write a new function or add an argument to `searchentries()` to say if we want to delete the matching entries. As it's a minor change we choose the latter.

This is option 3 in the list above.

### The new `searchentries()` function, with delete

The `searchentries()` function now takes an argument, `delete`, (and could probably now do with being renamed!). The `if()` statement within the loop now has two actions:

```
/*
 * List entries containing search string,
 * If the argument "delete" is non-zero, delete the entry.
 */
void searchentries(int delete) {
    int i;
    char searchstr[MAXLEN];

    if (delete)
        printf("Entries matching the following search string will be deleted\n");
    printf("Please enter the string to search for (case-sensitive)\n");
    if ( readoneline(searchstr, MAXLEN) == 0 )
        return;

    for (i = 0; i < MAXENTRIES; ++i)
        if ( entries[i][0] != '\0' && strstr(entries[i], searchstr) ) {
            printf("%s\n", entries[i]);
            if ( delete )
                entries[i][0] = '\0'; /* Mark as unused */
        }
}
```

### Checking for errors with `assert()`

A depressingly common problem is that something that "cannot be true" sometimes is due to a bug in the code.

The `assert()` macro, which is defined inside `assert.h`, has a simple and brutal function: if the expression in parentheses is false (zero) the program is killed there and then via the `abort` signal. This throws us back into the debugger, if we were running under the debugger, or writes a debug file if we were not. For example the line:

```
assert(0);
```

will always kill the program (but see below).

### Another example: combining `assert()` and `isfinite()`

In a numeric code we sometimes find variables set to Not a Number or Infinity. To check for this we can use the useful `isfinite()` macro, declared inside `math.h`. This checks to see if an expression is a "proper" floating-point number. If it is NaN or infinity it returns zero.

For example, if we had a problem where one or more of the elements of an array were infinity or Not a Number, we could write:

```
for(i = 0; i < N; ++i)
    assert(isfinite(x[i]));
```

### Turning off `assert()`

For obvious reasons, `assert()` is only really useful if we are running under the debugger. For equally obvious reasons we don't want to leave it enabled when we release the program to the people who are going to use it. Therefore `assert()` can be turned off by defining `NDEBUG` before including `assert.h`:

```
/* Turn off assert */
#define NDEBUG
#include <assert.h>
```

In this case the pre-processor replaces `assert()` with a space.

We shall use `assert()` in the final version of the code to check for NULL file pointers.

## The final iteration with save to file

We now have quite a usable program with both search and delete. All we have to do now is to add the ability to save the data to a file and to read it back again.

Regarding the former, our old friend `listentries()` can then be called with `stdout` to write to the screen and a file opened with `fopen()` to write to a file. The test to check that the file was not equal to NULL could be done either inside or outside of `listentries()`, for convenience we will do it inside.

### Reading from the saved file

In a similar manner, we already have a function `readoneline()` to read an entry from the keyboard using `stdin`, all we have to do is to add another argument to `readoneline()` and it can read from the saved file. All we then need is a simple function to loop over this stopping either when there are no more free entries to store the data in or when we have run out of data.

We could just add an extra argument to `readoneline()` but as it's a general utility function we are probably using it in other programs and it would be confusing. In classic C style we will create a second version of `readoneline()` with an "f" on the end of its name, `readonelinef()`, with the `FILE *` argument at the end, since it's unformatted. The relevant code snippet looks like:

This is option 4 in the list above.

```
int readoneline(char line[], int maxbytes) {
    return readonelinef(line, maxbytes, stdin);
}

int readonelinef(char line[], int maxbytes, FILE *file) {
    // rest of original readline() here with stdin replaced by "file"
```

The basic loop to read in the saved data looks like this:

```
/* Read data as long as we have space to store it */
for (i = 0; i < MAXENTRIES; ++i)
    if (readonelinef(entries[i], MAXLEN, saved) == 0)
        break; /* Out of input data */
```

This stops in one of two ways: we are out of space, the `for()` loop's control statement, or out of input data, the `break` statement.

## The final code

```
/*
 * Simple notebook program, one entry per line with search and
 * delete. For example, each line could contain a name and phone
 * number. Automatically saves entries to SAVEFILE, defined below.
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#define SAVEFILE "datafile.txt"

/*
 * The entries. If entries[i][0] == '\0' that means that entry is
 * unused.
 */
#define MAXLEN 128
#define MAXENTRIES 256
char entries[MAXENTRIES][MAXLEN]; /* All set to '\0' as it is external */

int readoneline(char line[], int maxbytes);
int readonelinef(char line[], int maxbytes, FILE *file);
void newentry();
void listentries(FILE *file);
void searchentries(int delete);
void readsaved(void);

int main() {
    int choice;
```

```

/* Read saved entries from file */
readsaved();

/* Event loop */
while ( 1 == 1 ) {
    printf("\nMenu:\n\n"
           "\t1. Add new entry\n"
           "\t2. List all entries\n"
           "\t3. Search entries\n"
           "\t4. Find and delete entries\n\n"
           "\t0. Quit\n");
    scanf("%d", &choice);

    switch (choice) {
    case 0:
        listentries(fopen(SAVEFILE, "w")); /* Save data */
        return 0;

    case 1:
        newentry();
        break;

    case 2:
        listentries(stdout);
        break;

    case 4:
        searchentries(1);
        break;

    case 3:
        searchentries(0);
        break;

    default:
        printf("\nUnknown option \"%d\"\n", choice);
    }
}

/*
 * Read in and add a new entry
 */
void newentry(void) {
    int i;

    for (i = 0; i < MAXENTRIES; ++i) /* Find an empty slot */
        if (entries[i][0] == '\0') {
            printf("Please type in the new entry\n");
            readoneline(entries[i], MAXLEN) ;
            return;
        }

    /* If we got here the list was full */
    printf("Sorry, there is no space for any new entries\n");
}

/*
 * Read in a line of text, looping until it has some non-spaces.
 * Trim the final '\n' from the end
 * Return 1 for success, zero for failure.
 * Two versions: one from stdin, one from a file
 * It is an error for file to be NULL in readonelinef()
 */
int readoneline(char line[], int maxbytes) {
    return readonelinef(line, maxbytes, stdin);
}

int readonelinef(char line[], int maxbytes, FILE *file) { while ( 1 == 1 ) {
    int i;

    assert(file != NULL);

```

```

if ( fgets(line, maxbytes, file) == NULL )
    return 0; /* Out of data */

/* Trim the final new-line character */
i = strlen(line) - 1;
if ( line[i] == '\n' )
    line[i] = '\0';

/* Look for a non-blank character. */
for ( i = 0; line[i] != '\0'; ++i )
    if ( isspace(line[i]) == 0 )
        return 1;
}
}

/*
 * List all entries.
 */
void listentries(FILE *out) {
    int i;

    if ( out == NULL ) {
        assert(0); /* Note to self: does nothing if not debugging */
        fprintf(stderr, "Cannot write to file\n");
        return;
    }

    for ( i = 0; i < MAXENTRIES; ++i )
        if ( entries[i][0] != '\0' )
            fprintf(out, "%s\n", entries[i]);
}

/*
 * List entries containing search string,
 * If the argument "delete" is non-zero, delete the entry.
 */
void searchentries(int delete) {
    int i;
    char searchstr[MAXLEN];

    if ( delete )
        printf("Entries matching the following search string will be deleted\n");
    printf("Please enter the string to search for (case-sensitive)\n");
    if ( readoneline(searchstr, MAXLEN) == 0 )
        return;

    for ( i = 0; i < MAXENTRIES; ++i )
        if ( entries[i][0] != '\0' && strstr(entries[i], searchstr) ) {
            printf("%s\n", entries[i]);
            if ( delete )
                entries[i][0] = '\0'; /* Mark as unused */
        }
}

/* Read in saved data */
void readsaved () {
    FILE *saved;
    int i;

    if ((saved = fopen(SAVEFILE, "r")) == NULL ) {
        fprintf(stderr, "No saved data found\n");
        return;
    }

    /* Read data as long as we have space to store it */
    for ( i = 0; i < MAXENTRIES; ++i )
        if ( readonelinef(entries[i], MAXLEN, saved) == 0 )
            break; /* Out of input data */

    fclose(saved);
}
}

```

## A few notes

First, we have used `#define` for the file name. On start-up the program immediately calls `readsaved()` to read any previously saved data. The call to `fopen()` will fail if this is the first time we have run the program but we handle this without a problem.

Similarly, when we exit we automatically write the data to the save file. Although the call to `fopen()` should not fail in this case, we have added the check to `listentries()` to handle this gracefully. (We could perhaps have added a warning.)

The net effect is that if we quit the program and restart, we still have our data from the last time we ran the program.

To add a little polish we would have the program sort the output alphabetically, ask confirmation before deleting an entry, etc.

In the next module we will see how to have a variable message length and number of messages.

---