

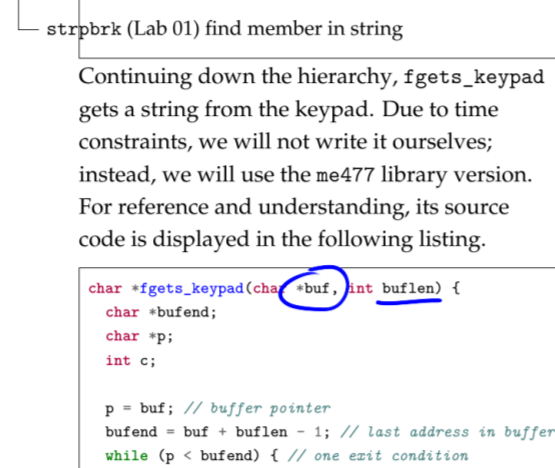
02.1 Lab Exercise: Keypad mid-level primitives

Objectives

- In this exercise you will gain experience with:
 - Code requirements for character I/O of a custom embedded computing application.
 - On-line debugging techniques.

Introduction

In Lab Exercise 01, we implemented a general-purpose function `double_a_is` that prompts the user to enter a floating-point value on the keypad, and returns the result to the calling program. That function calls the C functions `printf`, `atoi` and `fgets`, `keypad`. These functions, in turn, call other lower-level C library functions according to the following hierarchy. Functions provided by the `ncs77` library, `core C`, or the standard C library will be overwritten by those we write, which are shown in green.



Continuing down the hierarchy, `fgets_keypad` gets a string from the keypad. Due to time constraints, we will not write it ourselves; instead, we will use the `ncs77` library version. For reference and understanding, its source code is displayed in the following listing.

```
char *fgets_keypad(char buf, int buflen) {
    char *ptr;
    int n;

    ptr = buf; // buffer pointer
    buflen = buf + buflen - 1; // last address in buffer
    while (ptr < buflen && !feof(stdin)) {
        *ptr = getchar_keypad(); // get char from char array
        ptr++; // move pointer
        n++; // count char in buffer, increment pointer
    }
    *ptr = '\0'; // null-terminate string
    return ptr;
}
```

This function gets one keypad character at a time from the buffered `getchar_keypad` and writes them to the character array `buf` via the pointer provided as an argument of the function. In the exercise you will write the lower-level `getchar_keypad` function. This function acquires a single character from the keypad. It must eventually call the standard C function `getchar` that performs the same operations for the standard I/O device (if possible). You should review the `getchar` function in your C textbook.

In Lab Exercise 03, you will write the lower-level I/O functions `getkey` and `putchar_tod`.

Prerequisite program

Write the following functions and example (and debug) them before running them while connected to lab hardware.

Writing the buffered function `getchar_keypad`

The prototype of the `getchar_keypad` function should be as follows.

```
int getchar_keypad(void) // next char to type
```

```
void print_hi(void) {
    printf("hi\n");
}
```

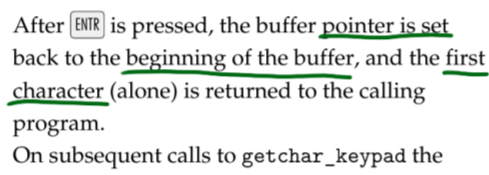
Each time `getchar_keypad` is called it returns a single character from the keypad; and it returns `EOF` (defined in `stdio.h`) when it encounters its representation of `EOF`. In the example below `getchar_keypad` is used to obtain a string of characters until `EOF` is reached. The characters are stored sequentially in a buffer pointed to by `point`.

```
while (1 && (getchar_keypad() != EOF)) {
    *point++ = *c;
}
```

There are two types of `getchar` functions in C. The first type, called an unbuffered `getchar`, simply returns the character to the calling program immediately after each keystroke. The second type, called a buffered `getchar`, collects the characters entered by the user in a temporary buffer. Pressing `EOF` causes the block of characters to be made available to the calling program. You will write a buffered `getchar_keypad` for the keypad.

The advantage of the buffered `getchar` is that the user can edit the characters in the buffer using the `ESC` key in the usual manner, before they are sent to the calling program. There is no possibility of editing with the unbuffered `getchar`.

You might wonder how a function designed to return only a single character could edit the whole buffer. This is accomplished by a simple and elegant means inside `getchar_keypad`. The key idea is to use a statically declared character buffer. In this way, the characters remain in the buffer in between calls to `getchar_keypad`. You will also need to statically declare a pointer to the buffer, and a variable (e.g. `n`) to keep count of the number of characters in the buffer. A schematic but it is easier, pointer, and count variable is shown below.



Here's how the buffered scheme should work. Whenever `getchar_keypad` is called either the buffer is empty or the buffer contains one or more characters.

The first time `getchar_keypad` is called, the buffer is empty, the count is zero (`n=0`), and the pointer is at the beginning of the buffer. The function enters a loop, filling the buffer and displaying the characters, one keystroke at a time, until the `ESC` key is pressed.

- Each time through the loop, it checks if the buffer is full. If it's not, it completes the following tasks:
- ✓ enter the current character into the buffer at the pointer's pointer.
 - ✓ increment the pointer.
 - ✓ increment the character count, and
 - ✓ print the character to the LCD.

After `ESC` is pressed, the buffer pointer is set back to the beginning of the buffer, and the first character (shown) is returned to the calling program.

On subsequent calls to `getchar_keypad` the buffer is not empty. For each call, the pointer is incremented, the count is decremented, and the character pointed to is returned to the calling program. This continues until the last character in the buffer is returned, and the pointer is returned to the beginning of the buffer. Once the buffer is empty, the next call to `getchar_keypad` begins the filling process again. Note: `getchar_keypad` should return `EOF` to represent the `ESC` key.

Putting these ideas together, algorithm pseudocode (so far) for a buffered `getchar_keypad` might look like that of Algorithm L.1, with

- `n` is the number of characters in the buffer.
- `buf` is a character array, of length `buf_len`.
- `p` is a pointer that points to the location in the buffer where the next character will be put or taken, and
- `chg` is the current character from `getkey`.

Now, suppose that the `ESC` is pressed while characters are being entered. The deleted character is effectively "removed" from the Algorithm L.1 buffered `getchar_keypad` pseudocode.

```
function getchar_keypad
    if n is 0 then
        point p a start of buf
        assign what getkey returns to chg
        while the chg is not EOF do
            if n < buf_len then
                assign chg to buf[p]
                increment n
            print chg to LCD with
            putchar_tod
            end if
            end while
        return chg
    else if
        point p a start of buf
    end if
    if n > 1 then // more than one character in
        decrement n
        return the pointer *p
    else if n is 1 then // one character in buffer
        decrement n
        return EOF
    end if
end function
```

buffer by decrementing both the buffer pointer `p` and the counter `n`. The deleted character is removed from the display by moving the cursor left one space, printing a space, and moving the cursor left one space again. What should happen if `ESC` is pressed before any characters have been entered (`n=0`)? Modify the pseudo code above (and your program) to include this "delete" functionality.

Writing the `main` function

Write a main function that tests your `getchar_keypad`. It should collect at least two separate strings using `fgets_keypad` (which calls `getchar_keypad`).

Table L.1: Left and right keys of `putchar_tod` map.

key	decimal code	symbol	esc seq	function
DEL	8	DEL		
ESC	10	ESC		
CL	45			clear display
←	48-57			cursor left, 1 space
→				cursor to start of Line-1
↑	91	UP		cursor to start of Line-2
↓	93	DN		

Background

To accomplish its task `getchar_keypad` must read characters from the keypad. The `getkey` function returns a single key code for each keystroke. Its prototype is as follows.

```
char *getkey(void)
```

A call to `getkey` might be: `ch = getkey();` Corresponding to each of the 16 keys of the keypad, the key code is shown in Table L.1. The symbols are `#def`ed in the header file `ncs77.h`.

In addition to getting keys, `getchar_keypad` must be able to print characters `,`, and decimal digits to the LCD screen. The `ncs77` library function `putchar_tod` should be used. Its prototype is as follows.

```
int putchar_tod(char c)
```

Both the input parameter and the returned value are the character to be sent to the display. The following are some examples of calls to `putchar_tod`.

```
int putchar_tod('a');
putchar_tod('a');
```

It prints the character corresponding to its argument on the LCD screen. The `putchar_tod` function uses the same escape sequences, as shown in Table L.1, as `printf_tod`, which we wrote in Lab Exercise 01.

Library Primitives

Test and debug your program.

Failure

The following guidance is provided for this week's lab exercise.

Compile-time integral constants

Often, we want to define a symbol that has a single integral value—an integer—throughout our program. Fortunately, C lets us do that many ways. Unfortunately, it can be hard to choose among them.

The primary ways are `#def` (macro), `enums` (enumerations), and `const` (ints). When choosing among them, our primary concerns are code readability, debuggability, and compile-time optimization.

The last of these means a compiler (or preprocessor before the compiler) can replace each instance of the symbol with its constant value (since it never changes). There are subtle differences in how each compiler works, but most of the time all three of our options yield reliable compile-time constants. However, `#def` (and `enums`) are the best guarantee (because it actually happens before compilation, via preprocessing), `enums` a close second, and `const` (a less respectable third).

In terms of debuggability, the rankings are probably best reversed; that is, in decreasing debuggability: `const`, `enums`, and `#def` (and `enums`). Macro (`#def`) tests are most difficult because the compiler can't usually give useful error codes related to them (since the compiler typically knows nothing of them due to preprocessing).

Readability is rather subjective, but `enums` are typically considered strong in this regard, especially with its automatic enumeration of symbols.

A way to demonstrate this is to show the same example, written these three ways. Let's define an integral value to each day of the week, then write a script that prints a value.

```
#include <stdio.h>
int main() {
    enum day {
        monday, tuesday, wednesday,
        thursday, friday, saturday,
    };
    enum day today = monday;
    enum day checkday = friday;

    int month = 1;
    printf("checkday is %d days", checkday - today);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    enum day {
        monday = 1,
        tuesday = 2,
        wednesday = 3,
        thursday = 4,
        friday = 5,
        saturday = 6,
        sunday = 7,
    };
    int month = 1;
    printf("checkday is %d days", checkday - today);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    const int monday = 1;
    const int tuesday = 2;
    const int wednesday = 3;
    const int thursday = 4;
    const int friday = 5;
    const int saturday = 6;
    const int sunday = 7;

    int month = 1;
    printf("checkday is %d days", checkday - today);
    return 0;
}
```

Preference among these three options is hotly debated, but it seems `enums` are the most readable and the "just right" option in terms of reliable compile-time integral constant replacement and debuggability. It is important to remember that `#def` (and `enums`) can be used for much more than integer replacement: function-like macros, for instance, are very useful.

Assigning to a pointer

The function `fgets_keypad`, the source for which is shown in the introduction to this lab, was used in Lab Exercise 01. Recall that in `double_a_is` we supplied an argument to `fgets_keypad` a character array (pointer) and its length. Instead of returning the string, the function wrote to the character array it was supplied—but remember: inside a C function arguments are assigned automatic variables. How does `fgets_keypad` assign to the array when it knows only a pointer to its first element? The secret sauce is to assign through a dereferenced pointer. Examine the source for `fgets_keypad` or consider the following example.

```
#include <stdio.h>
int main() {
    int month = 1;
    printf("month = %d\n", month);
    printf("month = %d\n", *p);
    printf("month = %d\n", **p);
    return 0;
}

void func(int *p) {
    *p = 2;
}
```

Note that, while this sort of structure is rare among higher-level programming languages, it is quite common in C. For instance, `fgets` and `gets` have this same feature.