Lab Exercise 02 Keypad mid-level primitives

Lab 02.1 Objectives

In this exercise you will gain experience with:

1. Code requirements for character I/O of a custom embedded computing application.
2. On-line debugging techniques.

Lab 02.2 Introduction

In Lab Exercise 01, we implemented a general-purpose function `double_in` 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_lcd` and `fgets_keypad`. These functions, in turn, call other lower-level C library functions according to the following hierarchy. Functions provided by the `me477` library, core C, or the standard C library will be overwritten by those we write, which are shown in **`green`**.

```
double_in (Lab 01) prompts LCD and returns keypad double
├─ fgets_keypad (Lab 02) gets string from keypad ← this lab!
│  └─ getchar_keypad (Lab 02) gets char from keypad ← this lab!
│     ├─ getkey (Lab 03) gets char from keypad
│     └─ putchar_lcd (Lab 03) prints char to LCD
├─ printf_lcd (Lab 01) prints string to LCD
│  ├─ putchar_lcd (Lab 03) prints char to LCD
│  └─ vsnprintf (Lab 01) assigns to formatted string
├─ sscanf (Lab 01) converts ASCII to binary
├─ strstr (Lab 01) find string in string
└─ strpbrk (Lab 01) find member in string
```

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 `me477` library version. For reference and understanding, its source code is displayed in the following listing.

```c
char *fgets_keypad(char *buf, int buflen) {
  char *bufend;
  char *p;
  int c;

  p = buf; // buffer pointer
  bufend = buf + buflen - 1; // last address in buffer
  while (p < bufend) { // one exit condition
    c = getchar_keypad(); // get char from char array
    if (c == EOF) // another exit condition
      break; // break while loop
    *p++ = c; // write to buffer, increment pointer
  }
  if(p == buf) return NULL; // just ENTR
  *p = '\0'; // write last character (NULL)
  return buf;
}
```

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 this lab exercise, you will write the lower-level `getchar_keypad` function. This function acquires a single character from the keypad. It must function identically to the standard C function `getchar` that performs the same operations for the standard I/O device (the console). You should review the `getchar` function in your C textbook.

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

Lab 02.3 Pre-laboratory preparation

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

Lab 02.3.1 Writing the buffered function `getchar_keypad`

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

```
int getchar_keypad(void) // void means no args
```

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 ENTR. 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 ( (ch=getchar_keypad()) != EOF ) {
  *point++ = ch;
}
```

unbuffered
buffered

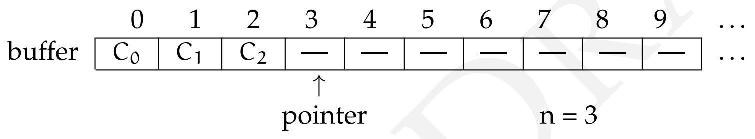
There are two types of `getchar` functions in C. The first type, called an *unbuffered* `getchar`, simply returns the character to the calling program immediately after each keystroke. The second type, called a *buffered* `getchar`, collects the characters entered by the user in a temporary buffer. Pressing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

editable input

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

static buffer

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 of the buffer, pointer, and count variable is shown, below.



Here's how the buffering 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 $\boxed{\text{ENTR}}$ key is pressed.

Each time through the loop, it checks if the buffer is full. If it's not, it completes the following tasks:

1. enter the current character into the buffer at the pointer's pointee,
2. increment the pointer,
3. increment the character count, and
4. print the character to the LCD.

After $\boxed{\text{ENTR}}$ is pressed, the buffer pointer is set back to the beginning of the buffer, and the first character (alone) 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 ENTR key.

Putting these ideas together, algorithm pseudocode (so far) for a buffered `getchar_keypad` might look like that of Algorithm 4, with

- `n` is the number of characters in the buffer,
- `buf` is a character array, of length `buf_len + 2`,
- `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`.

Algorithm 4 buffered `getchar_keypad` pseudocode

```
function GETCHAR_KEYPAD
    if n is 0 then                                          ▷ empty buffer!
        point p to start of buf
        assign what getkey returns to chg
        while the chg is not ENTR do
            if n < buf_len then
                assign chg to buf at p
                increment p
                increment n
                print chg to LCD with putchar_lcd
            end if
        end while
        increment n
        point p to start of buf
    end if
    if n > 1 then                            ▷ more than one character in buffer
        decrement n
        return the pointee *p
    else if n is 1 then                              ▷ one character in buffer
        decrement n
        return EOF
    end if
end function
```

Now, suppose that the ← is pressed while characters are being entered. The deleted character is effectively "removed" from the buffer by decrementing both the buffer pointer `p` and the counter `n`. The deleted character

Table 02.2: (left) keypad key codes and (right) `putchar_lcd` escape sequences.

key	decimal code	symbol
←	`8`	`DEL`
ENTR	`10`	`ENT`
-	`45`	
.	`46`	
0 – 9	`48 – 57`	
UP	`91`	`UP`
DWN	`93`	`DN`

esc seq	function
`\f`	clear display
`\b`	cursor left, 1 space
`\v`	cursor to start of Line-1
`\n`	cursor to start of Line-2

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 ← is pressed before any characters have been entered (`n==0`)? Modify the pseudo code above (and your program) to include this "delete" functionality.

Lab 02.3.2 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`).

Lab 02.4 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: `key = getkey();`

Corresponding to each of the 16 keys of the keypad, the key code is shown in Table 02.2. The symbols are `#define`d in the header file `me477.h`.

In addition to getting keys, `getchar_keypad` must be able to print characters `-`, `.`, and decimal digits to the LCD screen. The `me477` library function `putchar_lcd` should be used. Its prototype is as follows.

```
int putchar_lcd(int 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_lcd`.

```
ch = putchar_lcd('m');
putchar_lcd('\n');
```

It prints the character corresponding to its argument on the LCD screen.

The `putchar_lcd` function uses the same escape sequences, as shown in Table 02.2, as `printf_lcd`, which we wrote in Lab Exercise 01.

Lab 02.5 Laboratory Procedure

Test and debug your program.

Lab 02.6 Guidance

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

Lab 02.6.1 Compile-time integral constants

integral value 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 `#define`s (macros), **`enum`**s (enumerations), and **`const int`**s. 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 chances). There are subtle differences in how each compiler works, but most of the time all three of our options yield replaced compile-time constants. However, `#define`s are the best guarantee (because it actually happens before compilation, via *preprocessing*), **`enum`**s a close second, and **`const int`**s a respectable third.

preprocessing

In terms of debuggability, the rankings are probably best reversed; that is, in decreasing debuggability: **`const int`**s, **`enum`**s, and `#define`s. Macros (`#define`s) 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 **enum**s 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>
enum day {
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};
enum day today = monday;
enum day checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
#define sunday 0
#define monday 1
#define tuesday 2
#define wednesday 3
#define thursday 4
#define friday 5
#define saturday 6
#define today monday
#define checkout friday

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
const int sunday = 0;
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 today = monday;
const int checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

Preference among these three options is hotly debated, but it seems **enum**s 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 *#define*s can be used for much more than integer replacement: function-like macros, for instance, are very useful.

Lab 02.6.2 Assigning to a pointee

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_in` we supplied as arguments 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?

automatic variables

The secret sauce is to assign through a dereferenced pointer. Examine the source for `fgets_keypad` or consider the following example.

```
#include <stdio.h>
void foo(int * p);

int main() {
    static int x = 0;
    static int * p = &x;
    printf("before: %d\n",*p);
    foo(p);
    printf("after: %d",*p);
    return 0;
}

void foo(int * p) {
    *p = 3;
}
```

```
before: 0
after: 3
```

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.