operating systems lab - week 3: exercise

nicolo colombo

September 25, 2022

This lab is about memory, pointers, arrays, and strings. You will see in practice how pointers and arrays are very similar objects, how to pass them to functions, and use them to parse a general `stdin` input. You will learn how to handle a pointer array to make C programs accept command line arguments directly.

Set up

We suggest you edit, save, and compile the programs you write for this lab session in `CS2850Labs/week 3` a dedicated sub-folder of the directory you created for the first lab session of the term, on the teaching server, `linux.cim.rhul.ac.uk`. On the course Moodle page you can find more details about connecting to `linux.cim.rhul.ac.uk` and editing and compiling your code from the command line and debugging your programs using Valgrind.

1 Arrays

In this section, you will write a program that loads a set of integers entered by the user into an integer vector, prints all vector entries on separate lines, and computes the vector squared norm, $\|v\|^2 = \sum_{i=1}^{|v|} v_i^2$, using pointer arithmetics. The program input will be a series of nonnegative integers *separated by single spaces*, e.g.

```
1 12 123 1234
```

entered on the terminal by the user. As the memory to store strings and arrays cannot be allocated at runtime, you will need to print an error message if

i) the *length* of the input string or

ii) the *number* of nonnegative integers to be stored in the array

exceeds two pre-defined limits.
A run of your program should produce an output similar to

```
cim-ts-node-03$ ./a.out
enter nonnegative integers:
1 12 123 1234
input: 1 12 123 1234
a[0] = 1
a[1] = 12
a[2] = 123
a[3] = 1234
<a,a> = 1538030
```

where the second line is the user input. While the output of your program will change for different inputs, it is important that its format, e.g. spacing, capitalisation, and so on, matches the one in the example above. Start by creating a new file called `array.c` and saving it into this week's directory `CS2850Labs\Week3`. To write your program, follow the instructions in the next sections.

1.1 Parse the command line input

The function below reads the input character-by-character, loads the characters into a string, `s`, and stops reading when certain conditions are met.

```
int readLine(char *s, int MAX){                          1
  char c;                                                2
  int i = 0;                                             3
  while((c = getchar()) != '\n' && i<MAX){               4
    s[i++] = c;                                          5
  }                                                      6
  s[i]= '\0';                                            7
  return i;                                              8
}                                                        9
```

Read the code and answer the following questions.

- What is the return value of `readLine`?
- What is the meaning of the two `while` conditions? What is `MAX`?
- How is the input string passed to `main`?
- How can you rewrite Line 5 using pointer arithmetic, i.e. without squared-brackets notation?
- Modify the code of `readLine` so that the function
  1. prints the error message

     ```
     input too long!                                     1
     ```

  2. returns the *warning value* -1 if the user input contains more than `MAX` chars.
- What is the role of the statement on Line 7?

1.2 Print the input

Write a simple `main` where you call `printf` to print the accepted input. Before calling `readLine` you need to include the following lines

```
int MAX = 20;                                            1
char s[MAX];                                             2
```

where the first statement is for setting the maximum length (number of `char`) of the user input, i.e. the length of the *buffer*, and the second is for declaring an array of `char` of length `MAX`, i.e. to allocate the memory needed to store the content of the buffer.

1.3 Convert a block of `char` to the corresponding integer

So far, the input is a null-terminated array of `char` (thanks to Line 7 in `readLine`). To interpret a block of `char` as an integer you need to

- declare a *true* integer variable, e.g. `int n`,
- initialise a *true* integer, to 0 for each new block
- multiply each digit by the right power and add the value to `n`

You also need to ignore all possible *non-numerical* characters entered by the user, i.e. to discard all characters that are not in $\{'0', \ldots, '9'\}$. The conversion can be performed by calling the following function

```
int convertBlock(char *s, int *pos, int lenBlock){
  int n = 0, i = 0;
  while (i<lenBlock){
    char c = s[*pos + i];
    if((c —'0')<= 9 && (c — '0')>=0)
      n = n * 10 + (c — '0');
    i++;
    }
    *pos = *pos + lenBlock + 1;
  return n;
}
```

where

```
int getBlock(char *s, int *pos, int lenInput){
  int start = *pos;
  while (s[*pos] != ' ' && *pos < lenInput)
    *pos = *pos + 1;
  int len = *pos — start;
  *pos = start;
  return len;
}
```

Have a look at both codes and answer the following questions:

- Why can you check that a character is a digit through Line 5 of `convertBlock`?
- What is the role of `pos`? Why should you pass a pointer to the variable to both functions?
- Why do you need `+ 1` in Line 9? of `convertBlock`
- Why are there `*` in front of each occurrence of `pos` in `getBlock`?
- What is the meaning of the return value in `getBlock` and `convertBlock`?

1.4 Print the integer on the terminal

The following program includes a call to all functions defined above and is supposed to print on the terminal the *value* of the character blocks entered by the user.

```
int main(){
  int N = 4, MAX = 10;
  char s[...];
  printf("enter nonnegative integers:\n");
  int lenInput = readLine(..., MAX);
  if (... < 0) return —1;
  printf("input: %s\n", s);
  int pos = 0, lenBlock = ..., j = 0;
  while (... < lenInput && j++ < N && lenBlock){
    lenBlock = getBlock(..., ..., ...);
    printf("n=%d\n", convertBlock(..., ..., ...));
  }
  return 0;
}
```

Fill in the missing parts of the program so that it behaves as in the examples below.

```
enter nonnegative integers:
2 4 21
input: 2 4 21
n=2
n=4
n=21
```

```
enter nonnegative integers:
32 56
input: 32 56
n=32
n=56

enter nonnegative integers:
12  32 412 2
input too long!
```

1.5 Load the vector entries

Modify `main` given in the previous section so that the integers are *stored into an array* of `int` of maximum length `N` and the program prints an *error message* if the input contains more than `N` blocks. More precisely

- declare an array of integers of size `N`, e.g. by including the following statement in `main`

  ```
  int a[N];
  ```

- replace Line 11 with a statement to load `n` to the `j`-th entry of `a`
- include a condition to check if the number of blocks that have been read is too big and, in this case, i) print the error message "`too many entries!`" and

  ii) make the program exit

1.6 Print the vector and compute its norm

Finally, call

```
void printVector(int *a, int len){
  for (int k = 0; k<len; k++)
    printf("a[%d] = %d\n", k, *(a + k));
}

long computeNorm(int *a, int len){
  long norm = 0;
  for(int n = 0; n<len; n++)
    norm = norm + *(a + n) * *(a + n);
  return norm;
}
```

from `main` to i) print the entries of `a` and ii) compute and print its squared norm, i.e.

$$\|v\|^2 = \sum_{i=1}^{|v|} v_i^2 \tag{1}$$

Make sure that now a run of your program produces an output similar to

```
enter nonnegative integers:
1 23 45
input: 1 23 45
a[0] = 1
a[1] = 23
a[2] = 45
<a,a> = 2555

enter nonnegative integers:
1   3    123
input too long!
```

```
enter nonnegative integers:
1 2 3 4 5
input: 1 2 3 4 5
too many entries!
```

1.7 Test your program

To check your program, reduce the values of `MAX` and `N` and run it with short and very long inputs. The program should only parse allowed inputs and print error messages otherwise. Before doing that, run the executable with `valgrind` to see if you get error messages.

1.8 Save the final version of your program

In the Moodle revision tests, you will be asked to copy your full program into a sandbox and it will be tested automatically with a series of `stdin` inputs. To avoid unpleasant surprises,

- be sure you have saved the final version, i.e. the code you use wrote to complete Section 1.6, into the file called `array.c` and
- set the value of `MAX` to 20 and the value of `N` to 4
- check that the *format* of the output, including the error messages, empty spaces, exclamation marks, and newlines, is the same as in the examples of Section 1.6,

2 Command line arguments (optional)

Write a new version of `array.c` where the input is passed directly to `main`, i.e. let the definition of `main` start with

```
int main(int argc, char *argv[]){...}
```

The string array `argv` stores a variable number of arguments that the user can enter *separated by spaces* In this case, you can let the arguments be the character blocks parsed by the program of the previous section, e.g. a command line input such as

```
one1 two2 three3
```

would correspond to a string array `argv` of length 4 (remember that, by convention, nothing is stored in `argv[0]`). Command line arguments should be entered when the program is executed, e.g.

```
./a.out one1 two2 three3
```

In this case, there is no need to store the user input as a single string and you can access each block separately. Use the following *simplified version* of `convertBlock` to write a program that produces the same output as `array.c` (except for the error messages and the first three lines)

```
int convertBlock(char *s){
  int n = 0, i = 0;
  while (*(s+i)!='\0'){
    char c = *(s + i);
    if((c -'0')<= 9 && (c - '0')>=0)
      n = n * 10 + (c - '0');
    i++;
  }
  return n;
}
```