# Operating Systems Lab - Week 6: exercise
# - with answers

This lab is about dynamic memory allocation. You will learn to use `malloc` and `realloc` to allocate and re-allocate arrays and strings at run time. You will write

1. `sampling.c`, a program for sampling *without replacement* from $\{1, \ldots, N\}$ where $N$ is an integer entered by the user, and

2. `dynamicString.c`, a program for handling a *dynamic string* that grows as needed for storing what the user enters on the terminal.

You will be asked to implement these programs starting from their pseudocode.

## 1 Example

In this section, you will find an example of implementing a program given its *pseudocode*. Reuse `getInteger.c`, the code provided in this section, to implement `parseInput`, the input-parsing function that you need in `sampling.c`.

**Pseudocode** The pseudocode of `getInteger.c` is given in Algorithm 1. Pseudocode of `getInteger.c` **Input:** A maximum value for the parsed integer $n_{max}$ Define a macro `MAX` and set it to $n_{max}$ Declare a `char` variable, `c` Declare a `int` variable, `integer`, Let `integer` $= 0$ `c` $\neq \backslash n$ and `integer` $\leq$ `MAX` Let `c` $=$ `getchar`$()$ `c` $\in \{'0', '1', \ldots, '9'\}$ Let `integer` $= 10 *$ `integer` $+$ `c` $- '0'$ Do nothing `c` $\neq \backslash n$ Return $-1$ Return $0$ **Output:** -1 if the input is too large and 0 otherwise.

**A possible C implementation** Try to implement Algorithm 1 by yourself before looking at the following C code

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
int main() {
  char c;
  int integer = 0;
  while ((c = getchar())!= '\n' && integer<=MAX) {
    if (c >= '0' && c<= '9')
      integer = integer * 10 + (c - '0');
  }
  if (c != '\n') {
    return -1;
  }
  else{
    return 0;
  }
}
```

Connect each line in the C code above to the corresponding instruction in the pseudocode. Then answer the following questions:

- Are `c` and `integer` allocated in the *heap* or the *stack*?

  **Answer:** In the stack because their size is fixed at compile time.

- Why do you check if `integer` has exceeded the limit by checking whether `c` $\neq \backslash n$?

  **Answer:** Because this means that the `while` loop was exited before the end of the user input.

- Why do you need to subtract `'0'` when you update `integer`?

  **Answer:** As `c` is a character, its value is the corresponding ASCII number.

- Why, in this case, is it *safe* to declare `c` as a `char`?

  **Answer:**Because the exit keyword, `\n`, is a valid character. You may need to declare `c` as an integer if you use `EOF` instead.

## 2 Sampling without replacement

Random sampling $n$ elements from a given set *without replacement* is widely used in data science, e.g. if you need to choose $n$ random students from the student list `students.txt`. In this section, you will implement `sampling.c`, a C program that

- waits for the user to enter an integer $n$,
- dynamically allocates an array of integers, `a`, of size $n$,
- initializes the array so that `a`[`i`] $= i$, `i` $= 1, \dots, n$,
- samples without replacement *half of the entries* of `a`,
- prints on screen randomly selected entries.

**Pseudocode** The pseudocode of `sampling.c` is given in Algorithm 2. Pseudocode of `sampling.c` **Input:** A maximum value for the parsed integer $n_{max}$ Define a macro `MAX` and set it to $n_{max}$ Declare a `int` variable, `integer`, and initialize it to 0 Call `int parseInput(int *n)` with *the address of* `integer` as a parameter Let $b$ be the return value of `parseInput` $b = -1$ Do nothing and return $-1$ Let `sizeOfInt` be the size in bytes of an integer array Allocate a memory slot of `integer` $*$ `sizeOfInt` bytes in the *heap*, using `malloc` Let `a` be the pointer returned by `malloc` an cast it to a *pointer to* `int` Initialise the entries of `a` by letting $a = [0, \dots,$ `integer` $- 1]$ Call the sampling function `getSamples(int *vector, int length)` with parameters `a` and `integer` Free the allocated heap memory Return 0 **Output:** -1 if the input is too large and 0 otherwise

### Notes on Algorithm 2

- The parsing function

  ```
  int parseInput(int *n)
  ```

  should be defined in the same C file and obtained *by adapting* `getInteger.c` above. In particular, note that its parameter should be a pointer to `int` and its return value *is not the value of the parsed integer*.

- Implement your own *sampling function* or use

  ```
  void getSamples(int* v, int lv) {
    int i = lv/2;
    for (int j = 1; j <= i; j++) {
      int r = rand() % (lv - j + 1);
      int choice = *(v + r);
      *(v + r) = *(v + lv - j);
      *(v + lv - j) = choice;
    }
    printf("[");
    for (int j=1; j<=i; j++)
           printf(" %d ",*(v + lv  - j));
  ```

```
  printf("]\n");
}
```

If you decide to use the implementation above, ensure you fully understand how it works before copying it into `sampling.c`.

- Use

  ```
  void * malloc (size_t size)
  ```

  and

  ```
  void free (void *ptr)
  ```

  to allocate and free the memory in the heap. Check the details of their usage on this page of C online manual.

**Example.** When it runs, `sampling.c` should produce an output analogous to

```
./a.out
2
[ 1 ]
```

```
./a.out
13
[ 0  10  6  5  9  7 ]
```

```
./a.out
54
[ 19  2  37  34  43  10  48  3  11  1  30  35  20  6  46  44  26  18  16  42  38  8  7
     28  52  17  51 ]
```

**Answer:**

```
#include <stdio.h>
#include <stdlib.h>
#define MAX  1000
void getSamples(int* v, int lv) {
  int i = lv/2;
  for (int j = 1; j <= i; j++) {
    int r = rand() % (lv - j + 1);
    int choice = *(v + r);
    *(v + r) = *(v + lv - j);
    *(v + lv - j) = choice;
  }
  printf("[");
  for (int j=1; j<=i; j++)
         printf(" %d ",*(v + lv  - j));
  printf("]\n");
}
int parseInput(int *i) {
 char c;
 while ((c = getchar())!= '\n' && *i <= MAX) {
   if (c >= '0' && c<= '9')
     *i = *i * 10 + (c - '0');
 }
 if (c != '\n') {
   return -1;
 }
 return 0;
```

```c
}
int main() {
  int integer = 0;
  if (parseInput(&integer)< 0)
    return -1;
  int *a = malloc(sizeof(int) * integer);
  for (int j = 0; j < integer; j++)
        *(a + j)=j;
  getSamples(a, integer);
  free(a);
}
```

## 3 Dynamic string

A general limitation in the C codes you wrote in the past weeks was fixing the *maximum size* of the user input. In this section, you will write a C program, `dynamicString.c`, which creates and handles a string that grows to accommodate user inputs of any length. The idea is to store the characters in the heap and reallocate the string when more memory is needed.

**Pseudocode.** The pseudocode of dynamicString.c is given in Algorithm 3. Pseudocode of `dynamicString.c` **Input:** A buffer size $n_{buff}$ Define a macro `BUFFLENGTH` and set it to $n_{buff}$ Declare a `int` variable, `size`, and initialize it to `BUFFLENGTH` Declare a `int` variable, `nString`, and initialize it to 0 Declare a `int` variable, `c` Allocate a string of `BUFFLENGTH` characters in the *heap* $\texttt{c} \neq \texttt{EOF}$ $\texttt{nString} > \texttt{size} - 2$ Add memory for `BUFFLENGTH` *extra* characters to the string Add `BUFFLENGTH` to `size` Read a single character from the terminal and store it in `c` Copy `c` into the string at position `nString` Increment `nString` by 1 *Null-terminate* the string Call `void printString(char *string, int size)` to print the string and the size of the *allocated memory* on the terminal Free the string and exit **Output:** 0 if the execution reaches the end

### Notes on Algorithm 3.

- To reproduce the examples below, you need to set $n_{buff} = 10$, i.e. to include

  ```c
  #define BUFFLENGTH 10
  ```

  just below the *headers*. Run a few *sanity-check* of your program by changing the size of the buffer, e.g. try $n_{buff} = 3$ and $n_{buff} = 100$.

- Read single characters from the user input using `getchar`. To avoid compilation errors, include the call of `getchar` in the `while`-loop condition.

- Use

  ```c
  void * realloc (void *ptr, size_t newsize)
  ```

  to *re-allocate* the string when needed by writing in `main`

  ```c
  size = size + n * sizeof(char);
  s = realloc(s, size);
  ```

  where `s` is the pointer of the heap region currently allocated for storing the string and `size` the size of the new region (see this page of C online manual for more details). An equivalent but more explicit way of re-allocating the string is to call the following function, which only uses `malloc` and `free`:

  ```c
  char *increasesize(char *s, int *size, int nextra) {
    int newsize = *size + nextra * sizeof(char);
    char *temp = malloc(newsize);
    for (int i= 0; i< *size; i++)
      *(temp + i) = *(s + i);
  ```

```
  free(s);
  *size = newsize;
  return temp;
}
```

Try both versions to see if you notice any difference when you compile or run the programs.

- Use the following version of **printstring** to print the string in the required format and reproduce the output shown in the examples.

```
void printstring(char *string, int size) {
  printf("-------------------------\n");
  printf("%s\n", string);
  printf("-------------------------\n");
  printf("memory size: %d\n", size);
  printf("-------------------------\n");
}
```

**Example.** A run of your program should produce an output analogous to

```
./a.out
one
two
three four
5 and 6
seven eight nine        ten!
-------------------------
one
two
three four
5 and 6
seven eight nine        ten!

-------------------------
memory size: 70
-------------------------
```

The string contains a new line character **\n** as the last *valid* character. Execute your program with a text file as an input by using the redirection operator as explained in Week 5's lab sheet, e.g. run

```
ls -l > someText.txt
./a.out < someText.txt
-------------------------
total 48
-rwx------ 1 ugqm002 staff 16968 Oct 29 15:54 a.out
-rw------- 1 ugqm002 staff   718 Oct 29 15:53 dynamicString.c
drwx------ 2 ugqm002 staff   152 Oct 29 15:56 extras
-rw------- 1 ugqm002 staff   818 Oct 29 11:53 getInteger.c
-rw------- 1 ugqm002 staff   874 Oct 29 14:38 sampling.c
-rw------- 1 ugqm002 staff     0 Oct 29 15:56 someText.txt

-------------------------
memory size: 390
-------------------------
```

**Answer:**

```
#include <stdio.h>
#include <stdlib.h>
#define BUFFLENGTH 10
```

```

char *increaseSize(char *s, int *size, int nExtra) {
  int newSize = *size + nExtra * sizeof(char);
  char *temp = malloc(newSize);
  for (int i= 0; i< *size; i++)
    *(temp + i) = *(s + i);
  free(s);
  *size = newSize;
  return temp;
}

void printString(char *string, int size) {
  printf("——————————————————————————\n");
  printf("%s\n", string);
  printf("——————————————————————————\n");
  printf("memory size: %d\n", size);
  printf("——————————————————————————\n");
}
int main() {
  int size = BUFFLENGTH * sizeof(char);
  char *s = malloc(size);
  int i = 0, k = 0, c;
  while ((c = getchar()) != EOF) {
    if (i > BUFFLENGTH - 2) {
      s = increaseSize(s, &size, BUFFLENGTH);
      i = 0;
    }
    *(s + k) = c;
    i++;
    k++;
  }
  *(s + k) = '\0';
  printString(s, size);
  free(s);
  return 0;
}
```