Operating Systems Lab - Week 1: exercise - with answers

This lab is about getting started with the environment for this course and practising the basic concepts of C. You will write the code of a program that prints a fixed number of times the welcome string `hello world!`.

1 Getting started

1.1 Connect to the teaching server `linux.cim.rhul.ac.uk`

For pedagogical reasons, we ask you to write, compile, and run your programs on the Linux environment provided by the department, i.e. the RHUL Computer Science *teaching server*. You can connect to the teaching server with the available command line `ssh` client, e.g. `puTTY`, and use a command-line editor, e.g. emacs, vim, or nano. Otherwise, you can use the `NoMachine` graphical interface. In this case, you can open the terminal and the text editor on separate windows.

Depending on your OS, use the following instructions to connect to `linux.cim.rhul.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter `linux.cim.rhul.ac.uk` in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

1.2 Create a new directory

To see the content and navigate in your home directory use the UNIX commands: `ls`, `cd`, `..` Create a new directory, called `CS2850Labs`, by running

```
mkdir CS2850Labs
```

We suggest you use `CS2850Labs` to save and run all programs of this course. You can create a sub-directory of `CS2850Labs` called `weekI`, $\texttt{I} = 1, \ldots, 11$, with

```
mkdir weekI
```

Use `ls` to show the content of the current directory and `pwd` to print its path.

1.3 Create and edit a text file with a command line editor

Choose one of the following command-line editors: emacs, nano, or vim. From the terminal, you can create an empty file, `helloWorld.c`, and open it with the text at once by running

```
vim helloWorld.c
```

To enter characters, go to *insert mode* by typing `i`. Use `ESC` to go back to *command mode* and `:wq` (in command mode) to save an exit. Write something in the file and check that everything was saved correctly using

```
more helloWorld.c
```

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

2 Your first C program

2.1 Write the C code

Open `helloWorld.c` again, replace your name with the following C code

```
#include <stdio.h>
int main() {
  printf("hello, world\n");
}
```

save, and exit.

2.2 Compile and run your C code

Compile `helloWorld.c` by running

```
gcc -Wall -Werror -Wpedantic helloWorld.c
```

To see all compilation options, type `man gcc` in the terminal and scroll the page with the up and down arrows. To read more about the meaning of the flags `-Wall`, `-Werror`, and `-Wpedantic` have a look at the gcc online manual on your web browser.
If you now print on the screen the content of `week1`, you should find a new file, `a.out`, which is the executable of `helloWorld.c`. Try to open it with vim. What do you observe? Why does the content of the file look so strange? In command mode, type

```
:%!xxd -b
```

to see the binary in the right format.
To see what `helloWorld.c` does, execute the binary file, `a.out`, by running

```
./a.out
```

Check that your output is exactly as follows

```
hello, world
```

2.3 More *hello, world*

Add the following lines to your code (just below the first call to `printf`)

```
printf("hello");
printf(",");
printf(" ");
printf("world\n");
printf("hello, world\n hello, world\n  hello, world!\n");
```

and check that the output is

```
hello, world
hello, world
hello, world
 hello, world
  hello, world!
```

including the strange indentation and the exclamation mark. What happens if you swap the exclamation mark and the last *newline* symbol, `\n`?

Answer: You get

```
cim-ts-node-01$ ./a.out
hello, world
hello, world
hello, world
 hello,  world
  hello, world
!cim-ts-node-01$
```

2.4 Debugging

The free system `valgrind` contains powerful debugging tools for Linux programs. The Valgrind suite is already installed on `linux.cim.rhul.ac.uk` and we suggest you use it to detect possible bugs in the programs you write for these labs. To see what may be wrong with your program, run the following

```
valgrind ./a.out
```

and have a look at the messages printed on the terminal. For the moment, this may look unnecessary and the messages you get are quite trivial. But running such sanity checks will become more and more important in the following weeks. One of the hardest parts of learning C is to understand how to manage the memory allocated by a program and looking at the `valgrind` messages may save you hours of debugging work.

3 Control flow

The control structures `for` and `while` allows you to repeat an operation a given number of times. Their usage and syntax in C are similar to what you know from other programming languages, but we suggest you have a look at this C online manual for all the details.

3.1 Create loops with `for`

Copy the code given in Section 2.1 into a new file called `forHelloWorld.c`. Add the following *macro-substitution* instruction on Line 1

```
#define N 10
```

and write a `for`-loop to make the program print the string `hello, world` `N` times. See Section 4.11.2 of The C Programming Langaugefor more details about macro-substitution statements. A `for`-loop in C is specified by three quantities

- the *iterator*, which needs to be declared as an integer `int i` and initialised inside the `for`-loop arguments list
- the *stopping condition*, e.g. `i < 4`, which stops the iteration when *false*
- the *iteration step*, e.g. `i = i + 1`, which defines the increment of the iterator at each iteration

For example, a `for`-loop defined by

```
int i;
for (i = 3; i <= 6; i = i + 2) {
   doSomething(...);
}
```

will call the function `doSomething` 2 times.

Answer:

```
#include <stdio.h>
#define N 10
int main() {
  int i;
```

```
  for (i=0 i < N; i = i + 1)
    printf("hello, world\n");
}
```

3.2 `while`-loop (optional)

Have a look at the following program

```
#include <stdio.h>
#define N 10
int main() {
  int i;
  int sum = 0;
  for (i = 0; i < N; i = i + 2) {
    sum = sum + i;
    printf("%d + ", i);
  }
  sum = sum + i;
  printf("%d = %d\n", i, sum);
}
```

Can you predict what is the output on the terminal without compiling and running the program? The following C code uses a `while`-loop and an `if`-statement to produce an analogous output

```
#include <stdio.h>
#define N 10
int main() {
  int i = 0;
  int sum = 0;
  while (i < N) {
    if (i % 2 == 0) {
      sum = sum + i;
      printf("%d + ", i);
    }
    i++;
  }
  sum = sum + i;
  printf("%d = %d\n", i, sum);
}
```

Write a new version of both programs, so that the output of both becomes

```
1 + 3 + 5 + 7 + 9 = 25
```

Use `Valgrind` to see if your program runs correctly and the heap usage of your program.

Answer: With a `for`-loop

```
#include <stdio.h>
#define N 10
int main() {
  int i;
  int sum = 0;
  for (i = 1; i < N -1; i = i + 2) {
    sum = sum + i;
    printf("%d + ", i);
  }
  sum = sum + i;
  printf("%d = %d\n", i, sum);
}
```

and with a `while`-loop:

```c
#include <stdio.h>
#define N 10
int main() {
  int i = 0;
  int sum = 0;
  while (i < N — 1) {
    if (i%2==1) {
      sum = sum + i;
      printf("%d + ", i);
    }
    i++;
  }
  sum = sum + i;
  printf("%d = %d\n", i, sum);
}
```

Operating Systems Lab - Week 2: exercise - with answers

This lab is about C types, variables, and functions. You will see in practice how numbers and characters are represented in C and how you can define and call functions. You will also write programs that parse terminal input/output.

Set up

We suggest you edit, save, and compile the programs you write for this lab session in `CS2850Labs/week2` 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 Variables

Similarly to other programming languages, you can use variables of different *types* and different *storage classes*. This list of primitive data types contains all variables you can declare and use in C. In this section, you will write a program that prints on screen the size in bytes of the most common C types, i.e.

`char, int, unsigned int, float`

1.1 Integers

Start by declaring and initializing a variable of type `int` and print its value as in the following program

```
#include <stdio.h>                                  1
int main() {                                        2
  int a = 10;                                       3
  printf("a=%d\n", a);                              4
}                                                   5
```

Write the code above into a file called, `printInt.c` and compile and run it to check that it prints

```
a=10                                                1
```

on screen. You can modify and recompile `printInt.c` as suggested in the following questions:

1. What happens if the variable is declared outside `main`?

   Answer: The variable is automatically declared as a static variable but the output does not change.

2. What happens if you add a non-integer part in the initialisation of `a`, e.g. if you replace Line 3 with `a = 10.1234`?

   Answer: The non-integer part is truncated.

3. What happens if you initialised `a` with a *very large* value, e.g. if you replace Line 3 with `a = 2147483647` and `a = 2147483648`?

   Answer: The second initialization produces a compilation error/warning because the value is out of the `int` range

The following code produces the same output as the program above.

```
#include <stdio.h>
int a = 10;
void printValue() {
  printf("a=%d\n", a);
}
int main() {
  printValue();
}
```

Copy the new program into a new file, `printInt2.c`, compile it, and run it to check that its output on screen is indeed

```
a=10
```

Modify `printInt2.c` as suggested in the following questions:

1. What happens if the variable is declared *inside* `main`?

   Answer: You get a compilation error because the function does not know anything about `a`

2. What happens if you change the value of `a` inside main, e.g. if you add

   ```
   a = 11
   ```

   just before Line 7?

   Answer: The function see the new value because the variabvle is global

3. What happens if you change the value of `a` inside the definition of `printValue`, e.g. if you add

   ```
   a = 11
   ```

   just before Line 4?

   Answer: The function prints the updated value because the variable is global

1.2 `unsigned int`, `char`, and `float`

Write a modified version of `printInt.c` called `printTypes.c`, that prints

```
au=2147483648
ac=*
af=0.123456
```

on screen and where `au` is declared as an `unsigned int`, `ac` as a `char`, and `af` as `float`. To obtain the correct output you should also use the correct format identifiers, `%u` for `unsigned int`, `%c` for `char`, and `%f` for `float` in the call of `printf`. Have a look at this list of formatting symbols for more details.

Answer:

```
#include <stdio.h>
int a = 1234;
unsigned int au = 2147483648;
char ac = '*';
float af = 0.123456;
void printValue() {
  printf("a=%d\n", a);
  printf("au=%u\n", au);
  printf("ac=%c\n",  ac);
  printf("af=%f\n", af);
  printf("a=%e\n", (float) a);
```

```
}
int main() {
  printValue();
}
```

What happens if you use the `int` format, `%d`, instead of %u when you call `printf` in `printUnsigned.c`?

Answer: You get

```
au=—2147483648
```

Try also to print the value of the variables as an unsigned octal number, with `%o`, an unsigned hexadecimal number, with `%x`, and a floating-point number in exponential notation, with `%e`. Which conversions are allowed and which lead to a compilation error if the program is compiled using the `-Werror -Wall` flags?

Answer: A `float` cannot be printed as an octal or hexadecimal number, e.g. you get

```
format '%o' expects argument of type 'unsigned int', but argument 2 has type 'double'
format '%x' expects argument of type 'unsigned int', but argument 2 has type 'double'
```

and integers cannot be printed in the exponential notation, e.g. you get

```
format '%e' expects argument of type 'double', but argument 2 has type 'int'
format '%e' expects argument of type 'double', but argument 2 has type 'unsigned int'
```

Force the conversion by including a type *cast* in the *second argument* of `printf` as in the following example

```
#include <stdio.h>
int a = 1234;
void printValue() {
  printf("a=%e\n", (float) a);
}
int main() {
  printValue();
}
```

1.3 Sizes

The size of a given type can be obtained by calling the operator `sizeof(type)`, e.g.

```
unsigned long int sizeOfChar = sizeof(char);
```

idem with `int`, `unsigned int`, or `float`, or by letting the argument of `sizeof` be a pre-declared variable, e.g.

```
char a;
unsigned long int sizeOfChar = sizeof(a);
```

See Section A7.4.8 of The C Programming Langaugeor Section 3.11 of the GNU Online Manual for more details about the `sizeof` operator. Write a program, `sizeOfTypes.c`, that prints on the terminal the size in bytes of a `char`, an `int`, an `unsigned int`, and a `float`. Your program should print the size of each type on a different line, with each line being of the form

```
the size of a long int is 8 bytes
```

Note that the output of `sizeof` is an `unsigned long int`.

Answer:

```
#include <stdio.h>
int main() {
  printf("the size of a char is %lu bytes \n", sizeof(char));
  printf("the size of an int is %lu bytes \n", sizeof(int));
  printf("the size of an unsigned int is %lu bytes \n", sizeof(unsigned int));
```

```
  printf("the size of a float is %lu bytes \n", sizeof(float));
}
```

What happens if you use `sizeof` to get the memory size associated with an array? Modify your program so that it prints two extra lines reporting the size in bytes of a 10-dimensional array of `char` and `int` declared as

```
int vInt[10];
char vChar[10];
```

Answer: The size of an `int` or a `char` is multiplied by the number of items in the array.

1.4 Signed or unsigned `char` (optional)

The conversion of characters to integers depends on whether the compiler treats the variables of type `char` as signed or unsigned quantities. Try to understand if on your system they are signed or unsigned by looking at the error messages produced by `gcc -Wall -Werror -Wpedantic` when you compile a program such as

```
#include <stdio.h>
int main() {
  char a = 150;
  unsigned char b = 150;
  printf("a=%d and b=%d\n", a, b);
}
```

Answer: The compiler prints an error because 150 is out of range if `char` is a *signed variable of 1 byte* The conversion of a variable of type `int` into type `char` may cause some information to be lost. Copy, compile, and run the following program:

```
#include <stdio.h>
int main() {
  int a = 128;
  char c;
  c = a;
  a = c;
  printf("a=%d\n", a);
}
```

What do you observe? Can you explain why all problems disappear if initialise `a` with the value 127?

Answer: The value of `a` becomes -127. If `a` is initialised with 127 it keeps its value because 127 is within the range of a signed `char`

2 Terminal input/output: `getchar` and `putchar`

In this section, you will write a program that transforms all lower case letters of an input string into upper case letters. The standard library contains functions for reading or writing one character at a time:

1. `getchar()`, which reads the next input character and returns it, and
2. `putchar(c)`, which prints the character `c` on the terminal.

Read, and try to guess what the following program does

```
#include <stdio.h>
int main() {
  char c;
  while ((c = getchar())!= 'q') {
```

```
    putchar(c);
  }
}
```

Copy the code into a new file called `inputOutput.c`, compile, and run it to understand how it works by typing random character on the screen when the program starts.

Answer: A typical run produces

```
cim-ts-node-01$ ./a.out
2
2
a
a
sa
sa
e
e
v
v
aslkj
aslkj
sdmnsn  salkjsdd
sdmnsn  salkjsdd
q
cim-ts-node-01$
```

2.1 Change the *exit* keyword

When you run the program in `inputOutput.c`, the terminal shows a new empty line where you can type your text. The program execution is paused until you send a newline character, `\n`. Once all characters in the input have been processed the program stops again, waiting for more input. For exiting, you need to send an exit keyword that makes the `while`-loop condition false. Try to modify the program above so that:

- the program exits when you type on the space bar

  Answer: Replace `while`-loop condition with

  ```
  (c = getchar())!= ' ')
  ```

- the program exits when you send a newline character (`return`)

  Answer: Replace `while`-loop condition with

  ```
  (c = getchar())!= '\n')
  ```

- the program exits when you type `ctrl-d`

  Answer: Replace `while`-loop condition with

  ```
  (c = getchar())!= EOF)
  ```

The `ctrl-d` combination is a terminal shortcut for sending an *end of file* signal. In C, the end-of-file signal is represented by an `int`, called `EOF`, and quite often equal to -1, a value that is not taken by any *valid* `char`. Add a few lines to your code to check that `EOF` $= -1$ on your machine. In principle, you should be careful with comparing variables of type `char` to `EOF`, as the latter is defined as an `int`. We suggest you keep this in mind and have a look at Section 1.5.1 of The C Programming Langaugefor a discussion about `EOF` and `getchar()`. The easiest solution is to declare the variable used to store the output of `getchar()` as an `int`, i.e. to replace Line 3 with

```
int c;
```

Answer: Add these two lines to print the value of `EOF`

```
int i = EOF;
printf("i=%d\n", i);
```

2.2 Lower and upper cases

In the ASCII characters encoding, upper-case letters are ordered alphabetically from `A` to `Z` and followed by all lower-case letters, which are also ordered alphabetically from `a` to `z`, i.e.

$$\cdots, \quad \texttt{A}, \quad \texttt{B}, \quad \cdots, \texttt{Z}, \quad \texttt{a}, \quad \texttt{b}, \quad \cdots, \quad \texttt{z}, \quad \cdots$$

This fact can be exploited for converting upper-case letters into lower-case letters and *vice versa*. The size of the alphabet can also be computed by subtracting the value associated with `A` to the value associated `a`, e.g. through

```
int sizeOfAlphabet;
sizeOfAlphabet = 'a' - 'A';
```

Write a function, `int upper(int c){ ... }`, that checks if the input character, `c`, is a lower case letter and, in that case, transforms it into the corresponding upper case letter. `upper` can be a modified version of

```
int lower(int c) {
  if (c >= 'A' && c <= 'Z')
    return c + 'a' - 'A';
  else
    return c;
}
```

To see the effect of `lower`, replace `putchar(c)` with `putchar(lower(c))` in `inputOuput.c`.

Answer:

```
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c <= 'z')
    return c - ('a' - 'A');
  else
    return c;
}
int main() {
  int c;
  while ((c = getchar()) != -1)
    putchar(upper(c));
}
```

Finally, set the exit keyword of `inputOutput.c` to `EOF` and recompile it. Copy the following text

```
one two three
four five
six
```

into a file called `someText.txt` ans observe what happens when you run

```
./a.out < someText.txt
```

Answer: The output is

```
ONE TWO THREE
FOUR FIVE
SIX
```

because `EOF` is sent automatically at the end of the text file.

Operating Systems Lab - Week 3: exercise - with answers

This lab is about memory, pointers, arrays, and strings. You will see how pointers and arrays are similar objects and learn to use them in a C program that parses a general `stdin` input.

Set up

We suggest you edit, save, and compile the programs you write for this lab session in `CS2850Labs/week3` 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` , 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. You need to reproduce the exact format of the above, e.g. spacing and capitalization, to check your implementation in this week's revision test. Write your program by following the instructions in the next sections and save it in a file called `array.c` into this week's directory `CS2850Labs\Week3`.

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){
  char c;
  int i = 0;
  while((c = getchar()) != '\n' && i<MAX){
    s[i++] = c;
```

```
  }                                                                    6
  s[i]= '\0';                                                          7
  return i;                                                            8
}                                                                      9
```

Read the code and answer the following questions.

- What is the return value of `readLine`?

  Answer: The length of the string

- What is the meaning of the two `while` conditions? What is `MAX`?

  Answer: The program exits the loop when it reads a new-line char or the length of the input string exceeds the pre-defined limit `MAX`

- How is the input string passed to `main`?

  Answer: The chars loaded in the string can be seen by `main` because it is passed to the function through the pointer to its start

- How can you rewrite Line 5 using pointer arithmetic, i.e. without squared-brackets notation?

  Answer:

  ```
  *(s+(i++)) = c;                                                      1
  ```

- 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.

  Answer: Insert the following below Line 6

  ```
  if (i==MAX) {                                                        1
    printf("input too long!\n");                                       2
    return -1;                                                         3
  }                                                                    4
  ```

- What is the role of the statement on Line 7?

1.2 Print the input

Write a `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.

Answer:

```
int main(){
  int MAX = 10;
  char s[MAX];
  int i = readLine(s, MAX);
  printf("input: %s\n", s);
}
```

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`

Ignore all possible *non-numerical* characters entered by the user, i.e. 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`?

  Answer: Because in ASCII the numerical characters, like the alphabetic characters, are encoded sequentially, i.e.

  ```
  ascii(0)=48
  ascii(1)=49
  ascii(2)=50
  ascii(3)=51
  ascii(4)=52
  ascii(5)=53
  ascii(6)=54
  ascii(7)=55
  ascii(8)=56
  ascii(9)=57
  ```

- What is the role of `pos`? Why should you pass a pointer to the variable to both functions?

  Answer: `pos` keeps track of the current position on the input string. You need a pointer to keep track and updated its value over different calls of `getBlock` and `convertBlock`

- Why do you need `+ 1` in Line 9 of `convertBlock`?

  Answer: To skip the empty space between blocks

- Why are there `*` in front of each occurrence of `pos` in `getBlock`?

  Answer: To access the value of the pointer to `pos`

- What is the meaning of the return value in `getBlock` and `convertBlock`?

  Answer: The length of the block and the integer associated with the block

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!
```

Answer:

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

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`, and
- 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.

Answer: Replace Line 11 with

```
a[j++] = convertBlock(s, &pos, lenBlock);
```

and add

```
if (j == N){
    printf("too many entries!\n");
    return -1;
}
```

inside the loop.

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));
}
```

and

```
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 the 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;
}
```

Answer:

```
#include <stdio.h>
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;
}
long computeNorm(int *a, int len){
  long norm = 0;
  for(int n = 0; n<len; n++)
    norm = norm + *(a + n) * *(a + n);
  return norm;
}

int main(int argc, char *argv[]){
  int N = 5;
  int a[N];
  for (int i=1;i<argc;i++){
    a[i-1] = convertBlock(argv[i]);
    printf("a[%d]=%d\n", i-1, a[i-1]);
  }
  printf("<a, a>=%ld\n", computeNorm(a, argc - 1));
}
```

Answer:

3 Quiz solution codes

3.1 Your solution to this week's lab exercise

```
#include <stdio.h>
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;
}
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 + i + 1;
  return n;
}
int readLine(char *s, int MAX){
  char c;
  int i = 0;
  while((c = getchar()) != '\n' && i<MAX)
    *(s+(i++)) = c;
  if (i==MAX) {
    printf("input too long!\n");
    return —1;
  }
  s[i]= '\0';
  return i;
}
long computeNorm(int *a, int len){
  long norm = 0;
  for(int n = 0; n<len; n++)
    norm = norm + *(a + n) * *(a + n);
  return norm;
}
void printVector(int *a, int len){
  for (int k = 0; k<len; k++)
    printf("a[%d] = %d\n", k, *(a + k));
}
int main(){
  int N = 4;
  int a[N];
  int MAX = 20;
  char s[MAX];
  printf("enter nonnegative integers:\n");
  int lenInput = readLine(s, MAX);
  if (lenInput < 0) return —1;
  printf("input: %s\n", s);
  int pos = 0;
  int lenBlock = 1;
  int j  = 0;
  while (pos < lenInput && lenBlock){
    lenBlock = getBlock(s, &pos, lenInput);
    a[j++] = convertBlock(s, &pos, lenBlock);
    if (j == N){
      printf("too many entries!\n");
      return —1;
    }
  }
  printVector(a, j);
```

```
  long norm = computeNorm(a, j);
  printf("<a,a> = %ld\n", norm);
}
```

3.2 Sum of a string of digits

```
#include <stdio.h>
int sumAll(char *s, int MAX){
char c;
int len = 0;
  while((c = getchar())!= '0' && len<MAX)
    *(s+(len++)) = c;
  s[len]= '\0';
  int i = 0;
  int n = 0;
  while (i < len){
    c = *(s + i);
    if((c -'0')<= 9 && (c - '0')>=0)
    n = n + (c - '0');
    i++;
  }
  return n;
}
int main(){
  int MAX = 10;
  char s[MAX];
  int n = sumAll(s, MAX);
  printf("n=%d\n", n);
}
```

3.3 Int and float swap

```
#include <stdio.h>
void swap(int *i, float *v){
  float temp = *v;
  *v = *i;
  *i = temp;
}
int main(){
  int i = 1;
  float v = 2.3;
  swap(&i, &v);
  printf("i=%d and v=%f\n",  i, v);
}
```

3.4 Pointer arithmetics

```
#include <stdio.h>
int main(){
  int v[10];
  for (int i = 0; i < 10; i++)
    //v[i] = i * i;
    *(v+i) = i * i;

  for (int i = 1; i < 10; i++)
```

```
    printf("%d — %d = %d\n", v[i], v[i—1], v[i] — v[i—1]);
}
```

3.5 Strings are pointers

```
#include <stdio.h>
int main(){
  char s[20]="hello world!\n";
  *(s+11)='\0';
  printf("%s, ",s+6);
  printf("%s\n",s+6);
}
```

Operating Systems Lab - Week 4: exercise - with answers

This lab is about low-level input-output and processes. You will start to see why studying C is important from an OS perspective. You will practice with IO process control facilities. In particular, this exercise asks you to write

- a program that reads and writes files using `stdio.h` functions for *formatted IO* and
- a program that creates a given number of child processes using `fork`.

Try to reproduce the formatted output shown in the examples *exactly*, e.g. pay attention to all capitalization details and empty spaces.

1 Input-output

Write a program, called `inputOutput.c`, that

- takes two file names, e.g. `fileIn.txt` and `fileOut.txt`, as *command-line* arguments,
- copies what the user writes on the terminal after the program has started into the first file, `fileIn.txt`, and
- makes a *capitalised version* of the text saved in `fileIn.txt` into the second file, `fileOut.txt`.

1.1 Command line inputs

To make your program accept and parse command line arguments, you need the formalism mentioned in Week 3 lab introduction. You can find an example of how an input-dependent main in the last section of Week 3's lab exercise. Try to understand what the following program does

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 2) return -1;
  char *fileNameIn = argv[1];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  return 0;
}
```

How do you run the corresponding executable? Try different file names to see if the output is affected and if the file identifier, `fileno(fileHandleIn)`, changes over different runs.

Answer: The output is not affected because the name is stored in the struct as a string and referred to through a pointer to its first character. `fdIn` does not change over different runs. Make the program accept two files instead of one and print their identifiers and the size of their file handles. Why can't you use `printf` to print `fileHanldeIn` directly?

Answer:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
```

```
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  printf("fdOut=%d\n", fileno(fileHandleOut));
  printf("sizeof(fileHandleOut)=%lu\n", sizeof(*fileHandleOut));
  fclose(fileHandleIn);
  fclose(fileHandleOut);
  return 0;
}
```

`fileHandleIn` is a structure of type `FILE` and cannot be printed with a single call of `printf`.

1.2 Parse the user input with `scanf`

The following program uses the `stdio.h` function `fscanf` to print a capitalized version of the user input on `stdout`.

```
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c — 'a' + 'A';
  else
    return c;
}
int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}
int main() {
  char s[10];
  while (fscanf(stdin, "%10s", s)==1) {
    capitalise(s);
    fprintf(stdout, "s=%s", s);
  }
}
```

Compile and run the program to understand how `fscanf` works. Note that

- `fscanf` is triggered by both `'\n'` and `' '`,
- to avoid overflow or other memory problems, you need to specify the maximum number of characters to be stored in the buffer through the format specifier `%10s`, and
- `capitalize` returns the length of the string and the changes in `s` are not discarded when it returns.

Look at C online manualto see how to use its return value to exit the `while`-loop.

Answer: `fscanf` returns -1 when it reads a non-valid character, e.g. `EOF`.

1.3 Write the original input on `fileIn.txt`

Use `scanf` and the file handle associated with the first file to write the user input on `fileIn.txt`. Note that `stdin` in the program above is a file handle and, to write on an open file, you can use

```
fprintf(fileHandle, "%s\n", q);
```

where `fileHandle` is a *pointer* to the structure of type `FILE` associated with the open file.

Answer:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  printf("fdOut=%d\n", fileno(fileHandleOut));
  printf("sizeof(fileHandleOut)=%lu\n", sizeof(*fileHandleOut));
  char q[10];
  while(fscanf(stdin, "%10s", q) == 1)
        fprintf(fileHandleIn, "%s\n", q);
  fclose(fileHandleIn);
  fclose(fileHandleOut);
}
```

1.4 Capitalize and copy the content of `fileIn.txt` into `fileOut.txt`

Complete the following program using the correct file handles. The completed program should behave as described at the beginning of this section.

```c
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c - 'a' + 'A';
  else
    return c;
}
int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileIn = fopen(..., "w");
  char q[10];
  while(fscanf(..., "%10s", q) == 1) {
        fprintf(..., "%s", q);
  }
  fclose(...);
  ... = fopen(..., "r");
  FILE *fileOut = fopen(..., "w");
  while(fscanf(..., "%10s", q) == 1) {
    capitalise(q);
    fprintf(..., "%s", q);
  }
  fclose(...);
  fclose(...);
  return 0;
}
```

Note that the program does not print anything on screen. A run with the following user input.

```
one
Two

three four Five


and
s
i
x!
```

write

```
oneTwothreefourFiveandsix!
```

on `fileIn.txt` and

```
ONETWOTHREEFOURFIVEANDSIX!
```

on `fileOut.txt`.

Answer:

```c
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c — 'a' + 'A';
  else
    return c;
}
  int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}

int main(int argc, char *argv[]) {
  if (argc < 3) return —1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  char q[10];
  while(fscanf(stdin, "%10s", q) == 1)
    fprintf(fileHandleIn, "%s", q);
  fclose(fileHandleIn);
  fileHandleIn = fopen(fileNameIn, "r");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  while(fscanf(fileHandleIn, "%10s", q) == 1) {
    capitalise(q);
    fprintf(fileHandleOut, "%s", q);
  }
  fclose(fileHandleIn);
  fclose(fileHandleOut);
  return 0;
}
```

2 fork

Write a program, `nChildren.c`, where a parent process creates N child processes through `fork`, waits for them to complete a task, and exits. We suggest you use the following *standard library* functions:

1. `int printf(const char *format, ...)` defined in `stdio.h` and described in Section 12.12 of C online manual,
2. `pid_t getpid(void)` defined in `unistd.h` and described in Section 26.3 of C online manual,
3. `pid_t fork(void)` defined in `unistd.h` and described in Section 26.4 of C online manual,
4. `unsigned int sleep(int sec)` defined in `unistd.h` and described in Section 21.7 of C online manual,
5. `pid_t wait(int *status)` defined in `sys/wait.h` and described in Section 26.6 of C online manual, and
6. `int WEXITSTATUS(int status)` defined in `sys/wait.h` and described in Section 26.7 of C online manual.

Do not forget to include the corresponding headers (`stdio.h`, `unistd.h`, `wait.h`), write

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
```

at the very beginning of your code.

2.1 Command line argument

Again define `main` so that the program accept a single *character digit*, `N` as a command-line parameter, i.e. let `main` be

```
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  ...
}
```

What is `**argv`, why is `int main(int argc, char **argv)` equivalent to `int main(int argc, char *argv[])`? Why can you use `int N = *argv[1] - '0';` to convert the input into an integer?

Answer: `char **argv` is a pointer to pointer and is equivalent to a *string array* as it is initialized with a list of constant strings (the user input). `*argv[1]` is a `char` containing a numerical character, and can be converted into an integer by subtracting the right offset, i.e. the ASCII code of `'0'`.

2.2 Write a task function

The task of all children consists of

- printing the process identifier on the terminal using `printf` and `getpid` and
- sleeping for $n\%(N-1)$ seconds using `sleep`.

Your function should not return any value, i.e. you should declare it as `void`, and accept two parameters, the sleeping time and the process *label*. You can use the following structure

```
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", ..., ..., ...);
  sleep(...);
}
```

Answer:

```
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", j, getpid(), sec);
  sleep(sec);
}
```

2.3 Generate N children with fork

You can generate a given number of child processes with a loop. Add a return statement just after the children have performed their task to avoid an uncontrolled generation of child-of-child processes. Also, make your program print a message on the terminal when one of the children terminates using `printf`and the child's label $j = 1, \ldots, N$. For example, you can complete and add to your main the following lines

```
pid_t pid;
for (int j = 0; j < N; j++) {
  if ((pid = fork()) ...) {
    sleepingFunction(... , ...);
    printf("%dth child exits \n", ...);
    return j + 1;
  }
}
```

where N is the integer that you get from `argv`.

Answer:

```
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  int K = 3;
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if ((pid = fork())== 0) {
      //printf("%d", N/(2 * (j + 1)));
      sleepingFunction(N * (j % K), j + 1);
      printf("%dth child exits \n", j + 1);
      return j + 1;
    }
  }
```

2.4 Child-parent inter-process communication

Before exiting, the parent prints on the screen the order in which the children have terminated. The parents should also wait for all N children to terminate, which can be done by calling `wait` N times. Note that`pid_t wait(int *status)` returns the process identifier of the child that terminates and writes the return value of the child that terminates at the address passes as `status` parameter. To interpret the content of that address, you can call `WEXITSTATUS`, with the value stored at that address as an input. For example,

```
int status;
pidChild = wait(&status);
pidReturnValue = WEXITSTATUS(status);
```

where `pidReturnValue` is what we have called the *child label* above.

2.5 Print the order of arrival

Finally, the parent should print the order of the reaped children at the very end. To do this, save the return values of `wait` and `WEXITSTATUS` into two integer arrays and print their content just before the parent process terminates using

```
for (int k = 0; k < N; k++)
  printf("%dth child(pid=%d) exited %dth \n", orderVector[k], pidVector[k], k + 1);
```

Answer: Here is a possible version of the final program,

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", j, getpid(), sec);
  sleep(sec);
}
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  int K = 3;
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if ((pid = fork())== 0) {
      //printf("%d", N/(2 * (j + 1)));
      sleepingFunction(N * (j % K), j + 1);
      printf("%dth child exits \n", j + 1);
      return j + 1;
    }
  }
  int status;
  int pidVector[N];
  int orderVector[N];
  for (int k = 0; k < N; k++) {
    pidVector[k] = wait(&status);
    orderVector[k] = WEXITSTATUS(status);
  }
  for (int k = 0; k < N; k++)
    printf("%dth child(pid=%d) exited %dth \n", orderVector[k], pidVector[k], k + 1);
}
```

Example If $N = 3$ a run of the program should produce an output analogous to[1]

```
cim-ts-node-01$ ./a.out 3
1th child (pid=3691411) sleeps for 0 seconds
2th child (pid=3691412) sleeps for 1 seconds
1th child exits
3th child (pid=3691413) sleeps for 0 seconds
3th child exits
2th child exits
1th child(pid=3691411) exited 1th
3th child(pid=3691413) exited 2th
2th child(pid=3691412) exited 3th
```

[1] Of course, you should expect different values for the process identifiers.

Operating Systems Lab - Week 5: exercise
- with answers

In this lab, you will work with the UNIX shell and `bash` programming. In the first part, you will practice with simple functionalities of the UNIX shell, e.g. tree navigation commands and file inspection tools. In the second section, you will write simple `sh` programs to process a given text file.

1 The UNIX shell

In this section, you will get familiar with the most common commands of the UNIX shell. You will also learn how to extract specific lines of a text file with command line tools.

1.1 Basic commands

If you do not know commands such as `cd`, `ls`,`mkdir`, `cp`, `mv`, we suggest you read the corresponding pages of the *command-line manual*. The command-line manual is available directly from the shell, and you can open the page of a given `unix_command`, e.g. `ls`, by typing

```
man unix_command
```

in the terminal. For more info about the shell and its features, have a look at this week's slide or this online tutorial. We suggest you implement some examples proposed in the slides.

1.2 Background processes

In the first week of the term, you have seen how to *open and modify* a file with a command-line editor, e.g. emacs, nano, or vim. So far, you have been closing the editor to return to the shell, e.g. to recompile your files. The *ampersand* operator '`&`' allows you to keep the editor *open in the background* while you enter other command in the terminal, e.g. if you need to perform other *unrelated* tasks. Use your favorite editor, `editor_name`, to create a file, `students.txt` in your directory, copy the content of students.txt into it, save, and exit. Reopen `students.txt` in the background by entering the command

```
editor_name students.txt &
```

Now the shell outputs the "job number" and PID (process id) of the process running the editor but does not open the usual editor window. Try the following commands:

- `ps`, to see all running processes and corresponding PID,
- `fg process_name`, e.g. `fg vim`, to open the editor window,
- `ctrl-z` in the editor window, to return to the terminal,
- `kill PID`, to terminate the process from the terminal. [1]
- `ps`, to list the active processes in the shell,[2] and
- `killall process_name`, to kill all matching processes.

Use tab-completion not to type out entire directory names: after typing the first few characters of a directory or file, hit `tab` key to let the shell complete the name. If there's more than one match, you can press `tab` twice to see a list of matches.

[1] If, after trying to kill a process through `kill PID`, you are still seeing it, try to run `kill -SIGKILL PID` and then `editor_name students.txt &` or`kill -9 PID` to send a SIGKILL signal to the OS.

[2] You can use `ps` to obtain the PID of a given process and terminate it using `kill <PID>`.

1.3 Inspecting files

To see the content of `students.txt` without opening a text editor, you can use

- `cat`
- `more`
- `head` or `tail`

Check the command-line manual of all these commands and answer the following questions:

1. How do you stop scrolling the file and return to the shell when you use `more`?

   Answer: With the *quit* command `q`

2. What is the difference between

   ```
   more students.txt
   ```

   and

   ```
   cat students.txt
   ```

   Answer: `cat` shows the whole file without having to scroll.

3. What does the `-n` flag does in `cat`?

   Answer: `cat -n` shows the line numbers.

4. What is the output of the following command?

   ```
   cat students.txt students.txt
   ```

   Answer: The content of the file is shown twice.

5. How many lines of `students.txt` are shown if you type

   ```
   head students.txt
   ```

   and if you add the flag `-10`?

   Answer: 10 lines

6. What is the difference in the output of the two following command

   ```
   head —40 students.txt
   tail —40 students.txt
   ```

   Answer: The first shows the first 40 lines of the file and the second the last 40 lines.

For extracting *global info* from a file without inspecting its content directly you can use `wc` with various options. Check the manual to see how to use `wc` to print the *line count* of `students.txt`. Add an empty line at the end of the file. Does `wc` count it?

Answer: Enter the command

```
wc —l students.txt
```

which prints

```
203 students.txt
```

The last empty line is counted.

1.4 Filtering

You can sort the line of an input text file according to a specified criterion with `sort`. Try the following commands.

```
sort students.txt
sort —r students.txt
sort —t/ —k 2 students.txt.
```

What are the corresponding criteria for sorting the entries? How does the option `-t` work? And what is the difference between adding `-t" "` and `-t/`?

Answer: `-r` reverse the numerical order and `-t` allows you to specify a separator that you can use to focus on specific *fields* of the lines. In this case, you need to specify the priority field by adding `-k n` (for prioritizing the nth field). Another filtering command is `cut`, which also allows you to specify customized sorting strategies. Check the manual to see how you can use it to filter the information printed out from `students.txt`. Can you figure out how to show only the student names?

Answer:

```
cut —d / —f 2 students.txt
```

1.5 IO Redirection

Normally, command-line programs print to *standard output*, which is connected to the terminal by default. The IO redirection commands,

$$>, \quad <, \quad >>, \quad |$$

allow you to *read and write* data to disk or to communicate between different commands, i.e. processes, by connecting their standard input and standard output streams.

- $x>y$ redirects the output of x to file y,
- $x>>y$ redirects the output of x on file y without overwriting the file,
- $x<y$ uses the content of file y as input of the command x, and
- $x|y$ connects the standard output of command x to the standard input of command y.

Try to understand how the redirection operators work in practice by combining two or three of the UNIX commands mentioned in the previous sections as suggested below.

1. What is printed in the file `lsOut.txt` after running `ls > lsOut.txt`?

   Answer: The content of the current directory.

2. What happens if you run `ls -l >> lsOut.txt` three times?

   Answer: The file contains three times the content of the current directory.

3. What is the difference between running `wc students.txt` and `wc < students.txt`?

   Answer: The second command outputs only the stats about the file, without the file name.

4. Try to predict the output of the following command

   ```
   tail —10 students.txt | head —5
   ```

before running it.

Answer: The first five lines of the last 10 lines of the file.

5. Complete the second command below so that it produces the same output as the first one

```
sort students.txt|
head —5
sort ... students.txt| ... —5 | sort ...
```

Answer:

```
sort —r students.txt  tail -5 sort
```

Optional. Combine `cut`, `sort`, and the I/O redirection commands to print on a new file, `names.txt`, the student names (only their names) sorted alphabetically by first name.

Answer:

```
cut —d / —f 2 students.txt  sort
```

1.6 `grep`

To quickly inspect and filter text files you can also use `grep`, which allows you to print all lines that match a pattern. In particular, `grep` is a powerful tool when its argument is a *regular expression*. See wild cards list for a list of the wild cards you can use to build regular expressions in UNIX and the manual page of `grep` for further details about its syntax. Note that some regular expressions you can use with `grep` differ from the classical ones. Then answer the following questions:

1. What is the difference between the output of the two following commands

```
grep Candice students.txt
grep Ca[np] students.txt
```

Answer: The output of the second includes

```
1098/Caprice Cerrato/CS1801/CS1820/CS1830/CS1840/CS1860
```

2. How can you combine `grep` and `wc` to find the number of students taking CS1860?

Answer:

```
grep CS1860 students.txt  wc -l
```

3. How can you print the profile, i.e. the whole line, of the students who are *not* taking CS1890?

Answer:

```
grep —v CS1890 students.txt  wc -l
```

2 `sh` scripts

In this section, you combine UNIX command-line instructions into basic shell scripts. Before starting, have a look at Example 5.1, Example 5.2, and Example 5.3 for an explicit example of how to use a `for`-loop or a `while`-loop and write a program that performs a simple text filtering task.

2.1 Variables and inputs

Copy the following script into a new file, `myGrep.sh`,

```
#!/bin/sh
#myGrep.sh
IN=$2
OUT="out.txt"
PATTERN=$1
if test —f "$IN" ; then
        grep $PATTERN $IN > $OUT
        head —10 $OUT
fi
echo "file '$OUT' written"
```

and use `ls -l` to check its permissions. If you do not have the right to execute change the permission with

```
chmod u+x myGrep.sh
```

Then you can run it by typing

```
./myGrep.sh pattern file_name
```

where `file_name` should be `students.txt` and `pattern` is a standard `grep` search pattern, e.g.`100[13579]`. Can you write a single-line combination of the UNIX commands in `myGrep.sh` that produces the same output on the terminal, *except the last line* and without creating a *temporary* file `out.txt`?

Answer:

```
grep pattern fileIn.txt  head -10
```

2.2 ID filter

In a new file, `select.sh`, write a more refined version of `myGrep.sh` that accepts two integer parameters, `startID` and `endID` such that `startID` $\leq$ `endID`, and prints on the terminal the lines of `students.txt` corresponding to all students whose student ID is included in the range [`startID`, `endID`], i.e. all lines starting with an `ID` such that `startID` $\leq$ `ID` $\leq$ `endID`. The input file can be fixed and does not need to be passed as a parameter, i.e. you can write

```
IN="student.txt"
```

instead of `IN=$1` as in `myGrep.sh`. Start by completing the following `bash`-script

```
#!/bin/sh
# select.sh
IN="students.txt"
START=...
END=...
if [ "$#" —ne 2 ]; then
    echo "Usage: ... [startID] [endID]"
else
        LOOP=$START
        while [ $LOOP —le ... ]
        do
                grep ... $IN
                LOOP=`expr ...`
        done
fi
```

Answer:

```
#!/bin/sh
# select.sh
IN="students.txt"
START=$1
END=$2
FIRST=`grep $1 $IN  head -1`LAST=`grep 2IN  tail —1`
if [ "$#" —ne 2 ]; then
    echo "Usage: $0 [startID] [endID]"
else
        LOOP=$START
        while [ $LOOP —le $END ]
        do
                grep $LOOP $IN
                LOOP=`expr $LOOP + 1`
        done
fi
```

When you run it, your program should produce an output analog to

```
./select.sh 1181 1185
1181/Kiera Croslin/CS1801/CS1820/CS1890
1182/Kenny Mcclelland/CS1801/CS1820/CS1830
1183/Ilse Wheat/CS1801/CS1820/CS1830
1184/Gregorio Melia/CS1801/CS1820/CS1830
1185/Londa Stacker/CS1801/CS1820/CS1830
```

or

```
./select.sh 1181
Usage: ./select.sh [startID] [endID]
```

2.3 Print the student names

Make your program print the names of the first and last students in the range. For example, your new version should produce the following outputs

```
 ./filter.sh 1181 1185
first student=Kiera Croslin
last student=Londa Stacker
1181/Kiera Croslin/CS1801/CS1820/CS1890
1182/Kenny Mcclelland/CS1801/CS1820/CS1830
1183/Ilse Wheat/CS1801/CS1820/CS1830
1184/Gregorio Melia/CS1801/CS1820/CS1830
1185/Londa Stacker/CS1801/CS1820/CS1830
```

or

```
./select.sh 1181
Usage: ./select.sh [startID] [endID]
```

Start by inserting the following 2 lines at the right position in `select.sh` to print the first name

```
FIRST=`grep $1 $IN | head —1`
echo "first student=`echo "$FIRST" |cut —d / —f 2`"
```

Check that your program prints the first student's name as in the first line of the output above *only if* the program is launched with the expected arguments. Write two similar statements that print the name of the last student.

Answer: A possible script that works as requested is

```
#!/bin/sh
# select.sh
IN="students.txt"
START=$1
END=$2
if [ "$#" —ne 2 ]; then
    echo "Usage: $0 [startID] [endID]"
else
        FIRST=`grep $1 $IN | head —1`
        echo "first student=`echo "$FIRST" |cut —d / —f 2`"
        LAST=`grep $2 $IN | tail —1`
        echo "last student=`echo "$LAST" | cut —d / —f 2`"
        LOOP=$START
        while [ $LOOP —le $END ]
        do
                grep $LOOP $IN
                LOOP=`expr $LOOP + 1`
        done
fi
```

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

```
#include <stdio.h>                                                    1
#include <stdlib.h>                                                   2
#define MAX 1000                                                      3
int main() {                                                          4
  char c;                                                             5
  int integer = 0;                                                    6
  while ((c = getchar())!= '\n' && integer<=MAX) {                    7
    if (c >= '0' && c<= '9')                                          8
      integer = integer * 10 + (c — '0');                             9
  }                                                                   10
  if (c != '\n') {                                                    11
    return —1;                                                        12
  }                                                                   13
  else{                                                               14
    return 0;                                                         15
  }                                                                   16
}                                                                     17
```

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` $= 1, \ldots, n$,
- 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 $a = [0, \ldots,$ `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;
```

```
}
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* `c` $\neq$ `EOF` `nString` $>$ `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

  ```
  #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

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

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

  ```
  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`:

  ```
  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;
}
```

Operating Systems Lab - Week 7: exercise - with answers

This lab is about memory allocation, structures, and linked lists. In the first exercise, you learn to build a customized *allocator*. This may help with the dynamic allocation process and avoid memory leaks. In the second section, you will use a simply linked list to store words. Linked lists that can dynamically grow allow you to drop any constraints on the length of the user input.

1 Controlled memory allocations

Build a tool that prints on `stdout` the number of current *heap allocations*. The idea is to declare (in `main`) and update an *allocation counter*, `counter`, which is incremented by 1 every time you call `malloc` and is reduced by 1 every time you call `free`. More explicitly, write

- `heapAllocator`, a function that calls `malloc`, returns the pointer returned by `malloc`, and, if `malloc` returns a non-null pointer, increment the counter by 1, and
- `heapDeAllocator`, a function that calls `free` and reduces the counter by 1.

In both cases, the counter *is not* the return value of the functions.

1.1 Test program

As a test, you can use the C code of C example 7.2, available on Moodle. The program creates a linked list of `n` nodes storing one of the first `n` integers. Dynamic memory is needed because the user chooses the number of nodes at runtime. Check with Valgrind that the program does not leak memory when it runs and answer the following questions.

- In which order the nodes are printed on the screen?

  Answer: From the last node, which stores the integer `n`, to the first, storing 1.

- Is it possible to rewrite the entire code using the `dot-notation`, i.e. to remove all arrows "`->`" from the code?

  Answer: The code without arrows reads

```
#include <stdio.h>
#include<stdlib.h>
struct node{
  int val;
  struct node *next;
};
int main() {
  int n = getchar() - '0';
  struct node *head = NULL;
  for (int i=0; i < n; i++) {
    struct node *cur = malloc(sizeof(struct node));
    (*cur).val = (i+1);
    (*cur).next = head;
    head = cur;
  }
  struct node *cur = head;
  for (int i=0; i < n; i++) {
    printf("address node %d = %p\n", n-i, (void *) cur);
    printf("value node %d = %d\n", n-i, (*cur).val);
    printf("reference node %d = %p\n", n-i, (void *) (*cur).next);
```

```
    printf("-------------------------\n");
    if (cur) cur=(*cur).next;
  }
  for (int i=0; i < n; i++) {
    struct node *cur = head;
    head  = (*cur).next;
    free(cur);
  }
}
```

- What is the output of

```
printf("head->next->next->next->next->val=%d\n",
                          head->next->next->next->next->val);
```

if you insert it before and after the `for`-loop that prints the list?

Answer: If you set `n` = 6 you get `head->next->next->next->next->val=2`

- Remove the sanity check in the last line of the printing loops, i.e. replace

```
if (cur) cur=cur->next;
```

with

```
cur=cur->next;
```

What happens? Can you understand why you get an execution error and the program crashes?

Answer: When `cur` is `NULL` there is no `cur->next`

1.2 heapAllocator

The allocator is based on the memory-allocation function `malloc`. The function can be defined as

```
void *heapAllocator(int size, int *counter) {
  void *cur = malloc(size);
  if (cur)(*counter)++;
  return cur;
}
```

Answer the following questions:

1. Why do you need a pointer to an integer as a second argument?

   Answer: As `counter` is defined in `main`, we need the function not to discard the new values.

2. How does `main` know that the counter has been updated after the function has returned?

   Answer: Because `counter` is passed to the function through its address.

3. Would the function work for allocating a node of the integer list, as in C example 7.2, and a character array?

   Answer: Yes.

1.3 heapDeAllocator

The de-allocator is based on the memory de-allocation function `free`. The function can be defined as

```
void heapDeAllocator(void *p, int *counter) {
        if (p) (*counter)--;
        free(p);
}
```

Answer the following questions:

1. What is the return value of `heapDeAllocator`?

   Answer: Similarly to `free`, the function has no return value.

2. Can you explain what happens if you remove the parenthesis and the star in `(*counter)`?

   Answer: The counter is not updated because the function increments the pointer value but the changes are discarded when the function returns.

3. What happens if you call `heapDeAllocator` with a null pointer as a first argument?

   Answer: The function does not do anything.

1.4 Testing C example 7.2 for memory leaks

Use `heapAllocator` and `heapDeAllocator` defined above to check whether the program in C example 7.2 correctly frees all heap-allocated memory. Save the code of C example 7.2in a new file called `integerCheck.c` and modify the code as suggested below.

1. At the beginning of `main`, *declare* the counter, and initialize it to 0. Be sure that you set it before you start allocating memory dynamically.

2. *Replace* each call of `malloc` with a call to `heapAllocator`. Avoid compilation errors by passing the function's second argument in the right format.

3. *Replace* each call of `free` with a call to `heapDeAllocator`.

4. Print the *value of the counter* by adding the following line below each call of `heapAllocator` and `heapDeAllocator`,

   ```
   printf("counter=%d\n", counter);
   ```

5. In the `for` loop that prints the list, *comment out* all other calls to `printf`, except for the one printing the node value of the nodes, i.e. replace

   ```
   struct node *cur = head;
   for (int i=0; i < n; i++) {
     printf("address node %d = %p\n", n-i, (void *) cur);
     printf("value node %d = %d\n", n-i, cur->val);
     printf("reference node %d = %p\n", n-i, (void *) cur->next);
     printf("-------------------------\n");
     if (cur) cur=cur->next;
   }
   ```

   with

   ```
   struct node *cur = head;
   for (int i=0; i < n; i++) {
     printf("value node %d = %d\n", n-i, cur->val);
     if (cur) cur=cur->next;
   }
   ```

Check that your program compiles and runs without errors, that Valgrind produces no warning messages, and that your program has the following output (if you enter 3 in the terminal after it starts).

```
4
counter=1
counter=2
counter=3
counter=4
value node 4 = 4
value node 3 = 3
value node 2 = 2
value node 1 = 1
counter=3
counter=2
counter=1
counter=0
```

Answer:

```
#include <stdio.h>
#include<stdlib.h>
struct node{
        int val;
        struct node *next;
};
void *heapAllocator(int size, int *counter) {
        void *cur = malloc(size);
        if (cur)(*counter)++;
        return cur;
}
void heapDeAllocator(void *p, int *counter) {
        if (p) (*counter)--;
        free(p);
}
int main() {
        int counter = 0;
        int n = getchar() - '0';
        struct node *head = NULL;
        for (int i=0; i < n; i++) {
                struct node *cur = heapAllocator(sizeof(struct node), &counter);
                printf("counter=%d\n", counter);
                cur->val = (i+1) ;
                cur->next = head;
                head = cur;
        }
        struct node *cur = head;
        for (int i=0; i < n; i++) {
                printf("value node %d = %d\n", n-i, cur->val);
                if (cur) cur=cur->next;
        }
        for (int i=0; i < n; i++) {
                struct node *cur = head;
                head  = cur->next;
                heapDeAllocator(cur, &counter);
                printf("counter=%d\n", counter);
        }
}
```

2 A simply linked list of strings

In this section, you will write a program that creates a linked list to store a series of words entered by the user. The idea is

- to isolate single words of the input iteratively, by stopping reading characters when you reach `' '`,
- to allocate a new node of the list for each new word,
- to store the words and their length in the nodes, and
- to stop parsing the input when you reach a new line character.

You will need a series of subroutines given in Section 2.6. The structure of the main program is also given and you only need to complete a few statements. Save your code into a file called `linkedWords.c` and do not forget to check your code with Valgrind to see if it runs correctly.

2.1 Define a node

Avoid overflow problems in a structure definition, by fixing the maximum number of characters stored in a node. Let `MAX` be a macro defined at the beginning of the program as

```
#define MAX 10
```

and each node be an instance of

```
struct node {
    int length;
    char word[MAX];
    struct node *next;
};
```

where the `length` will store the string length, `word` the input word, and `next` the pointer to the next node.

2.2 Parsing the input

The input is processed through the function

```
{\tt int getWord(char *buf, int *end, int maxLength)}
```

that

- copies a single word from `stdin` into `buf`,
- sets `end` to 1 at the end of the `stdin` input, and
- returns the number of processed characters.

The number of processed characters is used to check if the *entire* word can be stored in the buffer and if a *valid* word has been found.

2.3 Creating a node

To store the string into a node, you need to

- allocate a new node using `malloc`,
- copy the string into the node's character array by calling,

  ```
  int copyString(char *in, char *out)
  ```

  which copies the content of `in` into `out` and returns the length of the string,

- stores the length of the string, obtained from

  ```
  int stringLength(char *s)
  ```

  that returns the number of characters in `s`, in the node's integer,
- link the current node to the *head* of the list, and
- move the head to the current node.

The memory allocation and the linking strategy are similar to the case of a list of integers.

2.4 Printing and freeing the list

To print the string iteratively, you can use

```
int printList(struct node *head, int n)
```

that prints the content of a list of `n` nodes, starting from the node pointed by `head`. To free the list, you can use

```
int freeList(struct node *head)
```

that frees the memory allocated for each node, starting from the node pointed by `head`.

2.5 Write `main`

As `main`, you can use the following template

```
int main() {
  char buf[MAX];
  struct node *head = ...
  struct node *cur = ...
  int end = 0;
  int count = 0;
  printf("enter words:\n");
  while (end == ...) {
    int j = getWord(...);
    if (j > 0) {
      buf[j]= ... ;
      cur = malloc(...);
      cur->next = ...;
      copyString(...);
      cur->length = stringLength(...);
      head = ...
      count = ...;
    }
  }
  int iPrint = printList(...);
  int iFree = freeList(...);
  printf("(count, iPrint, iFree)=(%d, %d, %d)\n", count, iPrint, iFree);
  return 0;
}
```

2.6 Input-parsing and string-handling

```
int stringLength(char *s) {
  int i = 0;
  while (s[i] != '\0') i++;
  return i;
}
```

```
int copyString(char *in, char *out) {
  int i = 0;
  while (in[i] != '\0') {
    out[i] = in[i];
    i++;
  }
  out[i]='\0';
  return i;
}

int getWord(char *buf, int *end, int maxLength) {
  int j = 0;
  char c = '\0';
  while (((c = getchar()) != ' ') && (c != '\n') && (j < maxLength))
    buf[j++]=c;
  if (j == maxLength) buf[j++] = c;
  if (c == '\n') *end = 1;
  return j;
}

int printList(struct node *head, int n) {
  struct node *iter = head;
  int i = 0;
  while (iter) {
    printf("%d-th node: %s (%d)\n", n - i, iter->word, iter->length);
    iter = iter->next;
    i++;
    }
    return i;
}

int freeList(struct node *head) {
  struct node *iter = head;
  int i = 0;
  while (head != NULL) {
    iter = (*head).next;
    free(head);
    head = iter;
    i++;
  }
  return i;
}
```

2.7 Example

A run of the program with user input,

```
one Two three FourFiveSix7Eight nine      Ten
```

should produce the following output.

```
./a.out
enter words:
one Two three FourFiveSix7Eight nine      Ten
7-th node: Ten (3)
6-th node: nine (4)
5-th node: x7Eight (7)
4-th node: FourFiveSi (10)
3-th node: three (5)
```

```
2—th node: Two (3)
1—th node: one (3)
(count, iPrint, iFree)=(7, 7, 7)
```

If you execute your program with Valgrind (and the same **stdin** input as above), you should not see any error or leaking message and print something similar to

```
valgrind ./a.out
==2239488== Memcheck, a memory error detector
==2239488== Copyright (C) 2002—2017, and GNU GPL'd, by Julian Seward et al.
==2239488== Using Valgrind—3.15.0 and LibVEX; rerun with —h for copyright info
==2239488== Command: ./a.out
==2239488==
enter words:
one Two three FourFiveSix7Eight nine      Ten
7—th node: Ten (3)
6—th node: nine (4)
5—th node: x7Eight (7)
4—th node: FourFiveSi (10)
3—th node: three (5)
2—th node: Two (3)
1—th node: one (3)
(count, iPrint, iFree)=(7, 7, 7)
==2239488==
==2239488== HEAP SUMMARY:
==2239488==     in use at exit: 0 bytes in 0 blocks
==2239488==   total heap usage: 9 allocs, 9 frees, 2,216 bytes allocated
==2239488==
==2239488== All heap blocks were freed —— no leaks are possible
==2239488==
==2239488== For lists of detected and suppressed errors, rerun with: —s
==2239488== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Answer:

```
int main() {
  char buf[MAX];
  struct node *head = NULL;
  struct node *cur = NULL;
  int end = 0;
  int count = 0;
  printf("enter words:\n");
  while (end == 0) {
    int j = getWord(buf, &end, MAX — 1);
    if (j > 0) {
      buf[j]='\0';
      cur = malloc(sizeof(struct node));
      cur—>next = head;
      copyString(buf, cur—>word);
            cur—>length = stringLength(buf);
      head = cur;
      count++;
    }
  }
  int iPrint = printList(head, count);
  int iFree = freeList(head);
  printf("(count, iPrint, iFree)=(%d, %d, %d)\n", count, iPrint, iFree);
  return 0;
}
```

Operating Systems Lab - Week 8: exercise
- with answers

In this lab, you will implement three programs that create a message-passing half-duplex pipe between processes that *have a common ancestor*:

1. `child2parent.c`, where the child sends formatted messages to its parent, as in `createPipe.c` ,
2. `parent2child.c`, where the parent sends formatted messages to its child, and
3. `child2child.c`, where the parent generates two children and the first child sends messages to the second child.

`createPipe.c` implements a *IPC channel* between a child and its parent by calling `pipe()` [1], which creates an *anonymous pipe* in the calling program. After calling `pipe()`, the program generates a child process with `fork()`. The *file descriptors* of the pipe ends are copied into the child address space and the two processes can communicate by *writing to* and *reading from* it. As *half-duplex pipes* are one-way message-passing channels, you must *choose* the direction of the information flow and close the *unused end of the pipe* in each process. For example, in `createPipe.c` , the parent can *only* read from and the child can only write into the pipe.

Three similar programs. Obtain `child2parent.c` by modifying `createPipe.c` . You only need a new implementation of the *reading* and *writing* subroutines. `parent2child.c` works in the opposite way. Assign the channel ends differently to make the parent write and the child read. Again the program structure will be very similar to `createPipe.c` . To write `child2child.c`, modify the main function because the parent will generate *two* children after creating the pipe. The first child will write messages into the pipe and the second will read the messages sent by the first. The parent will not interfere with the children's conversation and will close immediately both ends of the pipe.

Formatted messages. Instead of sending and receiving fixed hard-coded messages, the programs will convert the user `stdin` input into formatted messages, i.e. integers. In particular, the *writing process*

- reads a string from the terminal,
- separates the string into words, i.e. groups of characters ending with `' '`,
- processes the words one by one to check whether they contain *numerical characters*, i.e. `'0', '1', ..., '9'`,
- removes all *non-numerical* characters and converts the word into the corresponding integer, e.g. `one12two` will be converted into the integer 12, and
- send the obtained integer to the *reading process*, through the pipe.

The *reading process*, i.e. the child in `parent2child.c` and the second child in `child2child`

- reads the integers sent by the other process,
- computes their sum, and
- print the obtained value on the terminal.

Sending formatted messages through the pipe using the *high-level* I/O functions defined in `stdio.h` require referring to the pipe ends through pointers to their *file handle*, i.e. pointers to `struct`-objects of type `FILE`. You can obtain the pointers to the file handles of the pipe by calling

```
FILE *pReading = fdopen(fd[0], "r");
FILE *pWriting = fdopen(fd[1], "w");
```

[1] Include `unistd.h` in your program header.

1 child2parent.c

In this section, you will write a program, `child2parent.c`, where:

- an anonymous pipe is created by calling `pipe()`,
- a child process and a parent process are created by calling `fork()`,
- the child process converts an input string of words into a series of integers, e.g.

  ```
  ... one 1 two2 three34four and 5five6six7seven   ...
  ```

  will produce 0, 0, 1, 2, 34, 0, 567, and 0,
- the child sends the obtained integers (iteratively) to the parent as separate *formatted* messages through the pipe by calling

  ```
  fprintf(pWriting, "%d ",n)
  ```

  where `n` is the integer associated with the current word,
- the child sends a negative integer, e.g. `n` $= -1$ to tell the parents that the previous one was the last valid number obtained from the input,
- the parent reads the integers sent by the child by calling

  ```
  fscanf(pReading, "%d",n)
  ```

  ‘
- the parent computes the sum of the received messages by updating an integer variable `sum` through

  ```
  if (n >= 0) sum = sum + n;
  ```
- the parent prints the sum on the terminal before exiting.

1.1 Create a *high-level* pipe

Normally, when you create a pipe by calling `pipe()`, you refer to its ends through the corresponding *file descriptors*. In this section, you see how you can handle the pipe by using the corresponding *file handles*. As in `createPipe.c` , start by creating a 2-entry integer array declared as

```
int fd[2];
```

whose entries will be loaded with the file descriptors associated with the pipe's *reading* and *writing* ends(in this order). To create the new channel, write

```
pipe(fd);
```

exactly as in `createPipe.c` . Check whether the pipe has been created successfully by looking at the value of the return value of `pipe` and the entries of `fd`. To obtain the file handles associated with the file descriptors loaded in `fd`, write

```
FILE *pReading = fdopen(fd[0], "r");
FILE *pWriting = fdopen(fd[1], "w");
```

with `fdopen` being defined in `stdio.h`. See Section 13.4 of the GNU online manual for more details about `fdopen` and other similar functions. Note that you will need `pReading` and `pWriting` to use the *formatted I/O functions* defined in `stdio.h`, e.g. `fprintf` and `fscanf`.

1.2 Create a child process and close the unused ends of the pipe

The next step is to call `fork` and create a child process that will inherit both pointers to the pipe file handles, `pReading` and `pWriting`. In the child, who will be sending the messages, close the *reading* end of the pipe by calling

```
fclose(pReading);
```

In the parent, who will read the child's messages, close the *writing* end of the pipe by calling

```
fclose(pWriting);
```

Note that you should use `fclose` instead of `close` because `pReading` and `pWriting` are pointers to file handles. See Section 12.4 of the GNU online manual for more details about `fclose` and other similar functions.

1.3 Define the child and the parent processes

The child can now send messages to the parent through the pipe by writing on the *"file"* pointed by `pWriting`. To select the child process in your code, introduce an `if (!fork())-else` conditional block.

The child process. In the `if`-part of the conditional block, call the writing subroutine

```
int writeMessage(FILE *pf, char *author);
```

with the correct value of the first argument and the string `"the child"` as the second argument. As a writing subroutine, you can use

```
int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n')
      if (c<= '9' && c >= '0')
        n = n * 10 + c - '0';
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", -1);
  return 0;
}
```

When the writing subroutine returns the child has finished and can

- close the writing end of the pipe by calling

  ```
  fclose(pWriting);
  ```

- exit.

The parent process. In the `else`-part or the conditional block, call the writing subroutine

```
int readMessage(FILE *pf, char *author);
```

with the correct value of the first argument and the string `"the parent"` as the second argument. As a reading subroutine, you can use

```
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != -1) {
    fscanf(pf, "%d", &i);
    if (i != -1) {
```

```
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```

After the reading subroutine returns, the parent has *almost* finished its job and should

- print on `stdout` the sum of the received messages, i.e. the return value of `readMessage` by calling

  ```
  printf("sum=%d\n", sum);
  ```

- close the reading end of the pipe by calling

  ```
  fclose(pReading);
  ```

- and exit.

1.4 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
one 1 two 2 three3four 45five    s i x 6 85eigthyfive
the child writes to fd[1]: 0
the child writes to fd[1]: 1
the child writes to fd[1]: 0
the child writes to fd[1]: 2
the child writes to fd[1]: 3
the child writes to fd[1]: 45
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 6
the child writes to fd[1]: 85
the parent read from fd[0]: 0
the parent read from fd[0]: 1
the parent read from fd[0]: 0
the parent read from fd[0]: 2
the parent read from fd[0]: 3
the parent read from fd[0]: 45
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 6
the parent read from fd[0]: 85
sum=142
```

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);
int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  if (!fork()) {
    fclose(pReading);
    writeMessage(pWriting, "the child");
    fclose(pWriting);
    return 0;
  }
  else {
    fclose(pWriting);
    int sum  = readMessage(pReading, "the parent");
    printf("sum=%d\n", sum);
    fclose(pReading);
  }
}

int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c<= '9' && c >= '0')
        n = n * 10 + c — '0';
    }
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", —1);
  return 0;
}
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != —1) {
    fscanf(pf, "%d", &i);
    if (i != —1) {
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```

2 parent2child.c

Change a few details of `child2parent.c` so that

- the child reads and computes the sum of the messages sent by the parent and
- the parent converts the user input into integers and sends the obtained integers to the child

2.1 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
  one1 two23three 4fourfiveSix56
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 1
the child writes to fd[1]: 23
the child writes to fd[1]: 456
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 1
the parent read from fd[0]: 23
the parent read from fd[0]: 456
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
sum=480
```

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);
int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  if (!fork()) {
    fclose(pReading);
    writeMessage(pWriting, "the child");
    fclose(pWriting);
    return 0;
  }
  else {
    fclose(pWriting);
    int sum  = readMessage(pReading, "the parent");
    printf("sum=%d\n", sum);
    fclose(pReading);
  }
}

int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c<= '9' && c >= '0')
        n = n * 10 + c - '0';
```

```
    }
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", -1);
  return 0;
}
```

3 child2child.c

Change `child2parent.c` again so that

- the parent generates *two children*,
- the first child converts the user input into integers and sends the obtained integers to the child, and
- the second child reads and computes the sum of the messages sent by the first child.

3.1 Structure of `main`

You can use the following template for `main`.

```
int main() {
  ...
  for (int i = 0; i<2; i++) {
    if (i == 0) {
      if (!fork()) {
        ...
        return 0;
      }
    }else{
      if (!fork()) {
        ...
        return 0;
      }
    }
    wait(NULL);
  }
}
```

3.2 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
1    an d 2two three34 four    5fiveAndsix6
the writing child writes to fd[1]: 1
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 2
the writing child writes to fd[1]: 34
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
```

```
the writing child writes to fd[1]: 56
the reading child read from fd[0]: 1
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 2
the reading child read from fd[0]: 34
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 56
sum=93
```

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXCHARS 100
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);

int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  for (int i = 0; i<2; i++) {
    if (i == 0) {
      if (!fork()) {
        fclose(pReading);
        writeMessage(pWriting, "the writing child");
        fclose(pWriting);
        return 0;
    }
    }else{
      if (!fork()) {
        fclose(pWriting);
        int sum  = readMessage(pReading, "the reading child");
        printf("sum=%d\n", sum);
        fclose(pReading);
        return 0;
      }
    }
    wait(NULL);
  }
}
int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c >= '0' && c <= '9')
        n = n * 10 + c — '0';
    }
```

```
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d ", —1);
  return 0;
}
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != —1) {
    fscanf(pf, "%d", &i);
    if (i != —1) {
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```

Operating Systems Lab - Week 9: exercise - with answers

Implement a program that creates two concurrent threads to merge the content of two input files. Each thread reads from one of the files and writes on the *same* output file. The required Pthread functions are introduced in this week's slides. The main program waits for both threads to return, reads the content of the output file, and prints it on the terminal.

Similarly to the examples in the slides, start from a program that performs the two tasks sequentially, without creating any sub-procedure, and rewrite it as a multi-threads program. `0threadMerging.c`, the C code of the starting program is in Section 1. Sections 1 and 2 contain the instructions to rewrite it using one or two Pthreads. In Section 2, you implement

- the *sequential* setup, where the two tasks are performed *in series* by two independent Pthreads, i.e. the second Pthread starts when the first one has finished (in Section 2.1), and
- the *simultaneous* setup, where the two tasks are performed *in parallel* by two concurrent Pthread, i.e. the second Pthread starts immediately, without waiting for the first one to terminate (in Section 2.2).

The procedures open a given file, `input1.txt` or `input2.txt`, extract an integer from each *word* in the file, and print the integer on a given output file, `output.txt`. A word is a *group of characters between two single spaces*. The procedures transform it into an integer, as seen in previous labs. The extra spaces and the words containing only non-numerical characters would correspond to a 0 and should be ignored. To copy the integers into the output file, the procedures call the *formatted-I/O* function `fprintf`. The procedures call the same auxiliary function, `readIntegers`, with slightly different parameters, e.g. the name of the input file. The main program waits for both procedures to terminate and calls another function, `writeIntegers`, that i) reads the integers from the output file, ii) prints them on `stdout`, and iii) returns their sum. Finally, the program prints the sum on `stdout` and exits.

1 A program without Pthread

Copy `0threadMerging.c` and run it. Create the two input files by running

```
echo one1 two 2 three34four f5i5v5e and six=6 >input1.txt
echo 6six6 five54four t3h3r3ee and 2 and one1 > input2.txt
```

Then run the program to see what happens. Modify the two files to see how the output changes.

1.1 Making `readIntegers` Pthread-compatible

This is probably the hardest part of this exercise. Start by reading the code of the input-parsing function in `0threadMerging.c`, i.e.

```
int readIntegers(char *input, char *output,  char *threadName, char time) {
  char c = '\0';
  int nInt = 0;
  FILE *pf = fopen(input, "r");
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c — '0';
    }
    if (n) {
      nInt = nInt + 1;
      FILE *pfOut = fopen(output, "a");
      printf("%s writes %d \n", threadName, n);
```

```
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", threadName, nInt);
  fclose(pf);
  return nInt;
}
```

Try to understand the role of each argument and the meaning of the return value. Make `readIntegers` Pthread compatible by rewriting its argument and return value as *pointers to* `void`, define a new function declared as

```
void *PTreadIntegers(void *arguments)
```

Even if you pass it as a pointer to `void`, the argument should be a pointer to the following structure

```
struct pars {
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
  int time;
  int nInt;
};
```

where the first four members represent the four parameters of `readIntegers` and `n` its return value. To write `PTreadIntegers`, look at how `brian` and `dennis` are transformed into their Pthread compatible versions in this week's slides. You need to *cast* the function argument, `void * argument`, to a pointer to `struct args`. This is to access the structure members of the structure as usual. Otherwise, `argument` would remain a pointer to `void` and the compiler would produce an error if you write something like `argument->in`. More practically, start the definition of `PTreadIntegers` with

```
void *PTreadIntegers(void *par) {
  struct pars *p = par;
  ...
```

Below this first line, copy the code of `readIntegers` all occurrences of the parameters replaced by the corresponding structure members, e.g. `in` will become `p->in`.

Answer:

```
void *readIntegers(void *par) {
  struct pars *p = par;
  char c = '\0';
  FILE *pf = fopen(p->in, "r");
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < p->time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c - '0';
    }
    if (n) {
      p->nInt = p->nInt + 1;
      FILE *pfOut = fopen(p->out, "a");
      printf("%s writes %d \n", p->threadName, n);
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", p->threadName, p->nInt);
```

```
  fclose(pf);
  return NULL;
}
```

1.2 Rewriting `main`

Changing the function definition requires

- including the definition of `struct pars` on the top of the file,

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
  int time;
  int nInt;
};
```

- declaring two objects of type `struct pars` , in `main` and initialize them by calling

```
void initialisePar(struct pars *par, char *in, char *out, char *name, int time) {
  copyString(in, par->in);
  copyString(out, par->out);
  copyString(name, par->threadName);
  par->time = time;
  par->nInt = 0;
}
```

  where you can use

```
int copyString(char *in, char *out) {
  int n = 0;
  while (*(in + n) != '\0') {
    *(out + n) = *(in + n);
    n++;
  }
  *(out + n) = '\0';
  return n;
}
```

- changing the input-parsing function calls,
- replacing `writeIntegers` with

```
int writeIntegers(struct pars *t1, struct pars *t2) {
  int i = 0, sum = 0, n = 0;
  FILE *pf = fopen(t1->out, "r");
  while(n < t1->nInt + t2->nInt) {
    fscanf(pf, "%d", &i);
    sum = sum + i;
    printf("main reads %d\n", i);
    n++;
  }
  fclose(pf);
  return sum;
}
```

- including all redefined functions in the function declaration list on the top of the file.

Answer:

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
        char in[MAXCHARS];
        char out[MAXCHARS];
        char threadName[MAXCHARS];
        int time;
        int nInt;
};
int copyString(char *in, char *out);
void initialisePar(struct pars *par, char *in, char *out, char *name, int time);
void initialiseFile(char *out);
void *PTreadIntegers(void *parameters);
int writeIntegers(struct pars *t1, struct pars *t2);
int main() {
        struct pars par1, par2;
        initialiseFile("output.txt");
        initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
        initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
        PTreadIntegers(&par1);
        PTreadIntegers(&par2);
        int sum = writeIntegers(&par1, &par2);
        printf("sum=%d\n", sum);
}
```

2 A program with 2 Pthreads

The program can now call `pthread_create` and other functions defined in `pthread.h` and have two independent Pthreads execute the subroutine `PTreadIntegers`. In Sections 2.1 and 2.2, you implement the *sequential* and *concurrent* setups.

2.1 Pthreads in series

Two threads execute *sequentially* if the second is called just after the first one has terminated, i.e. if you wait for the first to terminate by calling `pthread_join` before starting the second. In this case, the Pthread commands should be

```
pthread_t t1, t2;
pthread_create(&t1, NULL, PTreadIntegers, &par1);
pthread_join(t1, NULL);
pthread_create(&t2, NULL, PTreadIntegers, &par2);
pthread_join(t2, NULL);
```

Answer:

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
```

```
  int time;
  int nInt;
};
int copyString(char *in, char *out);
void initialisePar(struct pars *par, char *in, char *out, char *name, int time);
void initialiseFile(char *out);
void *readIntegers(void *parameters);
int writeIntegers(struct pars *t1, struct pars *t2);

int main() {
  struct pars par1, par2;
  initialiseFile("output.txt");
  initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
  initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
  pthread_t t1;
  pthread_create(&t1, NULL, readIntegers, &par1);
  pthread_join(t1, NULL);
  pthread_create(&t1, NULL, readIntegers, &par2);
  pthread_join(t1, NULL);

  int sum = writeIntegers(&par1, &par2);
  printf("sum=%d\n", sum);
}
int copyString(char *in, char *out) {
  int n = 0;
  while (*(in + n) != '\0') {
    *(out + n) = *(in + n);
    n++;
  }
  *(out + n) = '\0';
  return n;
}

void initialisePar(struct pars *par, char *in, char *out, char *name, int time) {
  copyString(in, par->in);
  copyString(out, par->out);
  copyString(name, par->threadName);
  par->time = time;
  par->nInt = 0;
}
void initialiseFile(char *out) {
  FILE *pfOut = fopen(out, "w");
  fprintf(pfOut, "%s", "");
  fclose(pfOut);
}

void *readIntegers(void *par) {
  struct pars *p = par;
  int c = '\0';
  FILE *pf = fopen(p->in, "r");
  c = fgetc(pf);
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < p->time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c - '0';
    }
```

```
    if (n) {
      (p->nInt) = (p->nInt) + 1;
      FILE *pfOut = fopen(p->out, "a");
      printf("%s writes %d \n", p->threadName, n);
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", p->threadName, p->nInt);
  fclose(pf);
  return NULL;
}
int writeIntegers(struct pars *t1, struct pars *t2) {
  int i = 0, sum = 0, n = 0;
  FILE *pf = fopen(t1->out, "r");
  while(n < t1->nInt + t2->nInt) {
    fscanf(pf, "%d", &i);
    sum = sum + i;
    printf("main reads %d\n", i);
    n++;
  }
  fclose(pf);
  return sum;
}
```

2.2 Pthreads in parallel

Two threads will execute *concurrently* if the second is called just after the first has started. In this case, the order of the Pthread commands should be

```
pthread_t t1, t2;
pthread_create(&t1, NULL, readIntegers, &par1);
pthread_create(&t2, NULL, readIntegers, &par2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Answer:

```
int main() {
  struct pars par1, par2;
  initialiseFile("output.txt");
  initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
  initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, readIntegers, &par1);
  pthread_create(&t2, NULL, readIntegers, &par2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  int sum = writeIntegers(&par1, &par2);
  printf("sum=%d\n", sum);
}
```

2.3 Make the program *Pthread-safe*

As the threads are writing in the same file, `output.txt`, you must regulate their access to it. Introduce a *mutex* variable, `m`, declared outside all functions. In the function definition, use `pthread_mutex_lock(&m)` and `pthread_mutex_lock(&m)` to protect the writing statements, i.e. let

```
FILE *pfOut = fopen(p->out, "a");
printf("%s writes %d \n", p->threadName, n);
fprintf(pfOut, "%d ", n);
fclose(pfOut);
```

where `p` is the local pointer declared and initialized in the first line of `PTreadIntegers`.