C Programming Lab

1DT032: Advanced Computer Science Studies in Sweden
1DT086: Introduction to Studies in Embedded Systems
Uppsala University

September 3, 2015

Submitting Assignments, and Requirements for Passing this Lab

Solutions can be done individually or in groups of two students. All solutions have to be submitted via the following webpage:

`http://logicrunch.it.uu.se:4096/~wv/c-lab/`

The web service automatically checks whether uploaded code compiles correctly, and whether it passes a number of testcases.[1] If your solution is accepted you can continue with the next exercise, otherwise you will have to revise your solution and upload a fixed version.

Whenever you submit a solution, make sure that you have specified the name and personal number of all authors on the website.

In order to pass the lab, you have to submit correct solutions for at least 4 of the exercises. Submission opens on **Thursday September 3rd, 8:00** and closes at **Tuesday September 8th, 12:00.** The student or group submitting the highest number of correct solutions earns eternal glory (and a surprise present).

[1] Note: all submitted solutions are stored, and might manually be checked for correctness or signs of cheating at a later point.

The Lab Computers

You'll be working with the assignments in the Unix labs. If you are new to Unix or working from a terminal, the following table can be used as a quick reference. There are also many great resources online if you want to learn more.

`man <cmd>`	Shows a manual for the supplied command
`ls`	List files and directories in current directory
`pwd`	Show the working directory
`cd <dir>`	Go to the given directory
`cd ..`	Go back one directory
`mv <file1> <file2>`	Move file1 to file2
`rm <file>`	Remove a file
`rm -r <dir>`	Remove a directory and all files within it. Be careful, there is no undo button!
`cp <file1> <file2>`	Copy file1 to file2
`mkdir <dir>`	Create a directory
`Ctrl+C`	Key combination to abort most commands
`<command> &`	Run a command in the background
`gcc file.c -o output`	Compile `file.c` into executable file `output`
`gcc file.c -c -o file.o`	Compile `file.c` into object file `file.o`
`emacs file.c &`	Open the editor emacs and load `file.c`
`./program`	Run the executable file `program` in the current dir

Invoking the C Compiler

We will be using the GNU C Compiler (gcc). The manual for the compiler is quite long (15000+ lines), so don't worry about learning all of its features. To get you started, we go over the creation, compilation and execution of an example program.
First, we create a new directory to work in and step into that directory:

```
~]$ mkdir example
~]$ cd example
~/example]$
```

Then we create a C program file called `example.c`

```
~/example]$ emacs example.c
```

The file is edited to look as shown below:

```
/*
 * Author: Jonas Flodin
 */

#include <stdio.h>
```

```
int main(int argc, char ** argv){
  printf("This text is printed to the screen\n");
  return 0;
}
```

The file is saved by pressing **Ctrl+X** followed by **Ctrl+S**. Then we exit emacs by pressing **Ctrl+X** followed by **Ctrl+C**. (If you accidentally press some other command, you can abort it by pressing **Ctrl+G**). It should be noted that there are many editors other than emacs which you may prefer; vim, pico, gedit and nedit to name a few. You are of course welcome to use any editor of your choice.
We compile the program into an executable using gcc:

```
~/example]$ gcc -Wall example.c -o executable
```

The flag `-Wall` tells gcc to warn us about possible errors or design flaws that it can discover. It is a good habit to use this flag. You may also consider the `-std=c99` flag, which enables some newer additions to the C language, e.g., declaration of variables in the for loop header. Finally we list the files to see that some output has been produced and then we run the executable file.

```
~/example]$ ls
example.c  executable
~/example]$ ./executable
This text is printed to the screen
~/example]$
```

Now we've created and executed a program.

Makefiles and Compilation Units

In practice, C programs are usually split into multiple files, compiled separately, and in the end linked together to form a single executable. This way it is only necessary to recompile the modified files when changes are made to the program. Our example can be split into two files, one defining a function printing to the console (`printfun.c`), and one containing the main function (`example2.c`):

```
// example2.c

#include "printfun.h"

int main(int argc, char ** argv){
  printfun();
  return 0;
}
```

```c
// printfun.c

#include <stdio.h>

/**
 * Function implementation
 */
void printfun() {
  printf("This text is printed to the screen\n");
}
```

The function defined in `printfun.c` has to be declared in a separate *header* file, which is `$included` in `example2.c`:

```c
// printfun.h

/**
 * Function prototype
 */
void printfun();
```

Compilation now requires three calls to gcc: two to compile the C files to object files (with option `-c`), and one for the final linking to an executable:

```
~/example]$ gcc -Wall example2.c -c -o example2.o
~/example]$ gcc -Wall printfun.c -c -o printfun.o
~/example]$ gcc -Wall example2.o printfun.o -o example2
~/example]$ ./example2
This text is printed to the screen
~/example]$
```

Compilation is usually automated with the help of a build system, for instance with the tool `make`. The compilation steps are defined in a file named `Makefile`; for each file that is supposed to be generated through compilation or linking (here, `example2.o`, `printfun.o`, and `example2`) a set of commands is specified that produces the file. The `Makefile` also defines dependencies between source and target files: for instance, `example2.o` depends on `example2.c` and `printfun.h`, whenever any of the latter two files change, also `example2.o` has to be recomputed:

```
# Makefile

all: example2

example2.o: example2.c printfun.h
	gcc -Wall example2.c -c -o example2.o

printfun.o: printfun.c
	gcc -Wall printfun.c -c -o printfun.o

example2: example2.o printfun.o
	gcc -Wall example2.o printfun.o -o example2
```

Compilation is now started with a simple call to `make`:

```
~/example]$ make
gcc -Wall example2.c -c -o example2.o
gcc -Wall printfun.c -c -o printfun.o
gcc -Wall example2.o printfun.o -o example2
~/example]$ ./example2
This text is printed to the screen
~/example]$
```

Exercises

Now that we know how to create and run a program, we move on to the exercises. The solutions to (most of) the exercises will be provided after the end of the lab. Please refer to the slides for more information on C programming.

Exercise 1 Output

In the introductory program we include the library `stdio.h`, which contains the function `printf`. This function is used to produce output in the form of characters that are printed in the terminal. In its simplest form, the function is called with a string as argument:

```
printf("This text is printed to the screen\n");
```

That is great, but we want our programs to output more than just the fixed strings that the programmer writes in the program. To print the contents of variables, we add *format specifiers* to the string and add the variables we want to print as arguments:

```
int number;
char letter;
printf("%d is an integer and %c is a character\n", number, letter);
```

Different types of variables have different specifiers, all starting with a percentage sign. Common specifiers are `%d` for integers, `%f` for floats, `%c` for characters and `%s` for strings. To output a percentage sign, we use `%%`.
Write a function that outputs:

a) The string: `One half is 50%`

b) two integers and their difference.

c) two floats and the result of dividing one with the other

Write a main function that calls your other functions. The output has to be as follows:

```
[.../exercise]$ ./e1
One half is 50%
The difference between 10 and 3 is 7
1.000000 / 3.000000 is 0.333333
[.../exercise]$
```

Exercise 2 Input

For input we use the function `scanf`, also from the library `stdio.h`. The `scanf` function takes a format string followed by references to where the input should be stored. Example that reads an integer to a variable:

```
int number;
scanf("%d", &number);
```

Notice that the `&` character in front of the varable name. It means that the variable is passed as *reference* to `scanf`. It allows `scanf` to update the value of the variable. If `&` is not there, the program would likely crash at that point. When reading a string, the `&` sign can be omitted:

```
char my_variable[100];
scanf("%s", my_variable);
```

Write functions that:

a) asks for two integers and outputs them and their sum.

b) asks for two floats and outputs their product.

c) asks for a word and prints it twice on the same row.

Write a main function that calls your other functions. The output has to be as follows:

```
[.../exercise]$ ./e2
Give two integers: 12 5
You entered 12 and 5, their sum is: 17
Give two floats: 3.14 2
You entered 3.140000 and 2.000000, their product is: 6.280000
Give a word: Yey!
Yey! Yey!
[.../exercise]$
```

Exercise 3 Conditionals

If-else statements are used to make a program behave differently depending on the program state or user input. As an example, one can use if-statements to make sure that input is sane before performing an operation

```
int a;
int b;
...
if(b == 0){
  printf("Error: Divide by zero!\n");
  // Code for error handling.
  ...
}
else{
  printf("Division evaluates to: %d\n", a/b);
}
```

Write functions that:

a) ask for an integer and output whether the entered number is zero or not.

b) ask for two floats and outputs the largest of the inputs

c) ask for an integer and, if the number is divisible by two, divides it by two, otherwise multiplies it by three and output the result. Here, the modulo operator `%` is useful.

d) ask for three integers and output whether any of them are equal. Use only one if-else-statement

Write a main function that calls your other functions. The output has to be as follows:

```
[.../exercise]$ ./e3
Give an integer: 12
The number you entered does not equal zero
Give two floats: 13.4 20
20.000000 is the largest
Give an integer: 14
Result is: 7
Give three integers: 1 13 1
Some numbers are equal
[.../exercise]$ ./e3
Give an integer: 0
The number you entered equals zero
Give two floats: 13.2 -150
13.200000 is the largest
Give an integer: 7
Result is: 21
Give three integers: 2 5 13
All are unique
[.../exercise]$
```

Exercise 4 Loops

Loops are used to execute a statement or a block of code multiple times. A loop will continue to execute as long as the loop condition is satisfied. These two example loops will print the numbers 1 to 10 on one line and then 11 to 20 on the next line:

```
int i,j;
i = 1;
while(i < 11){
```

```
  printf("%d  ", i);
  i=i+1;
}
printf("\n");
for(j=11;j<=20;j++){
  printf("%d ", j);
}
printf("\n");
```

Write functions that:

a) print all even numbers between 0 and 40.

b) print all the numbers between 1 and 100, with 10 numbers on each line. Use two `for` loops. All columns should be aligned.

c) ask for a number than prints the number squared. This repeats until the 0 is entered.

Write a main function that calls your other functions. The output has to be as follows:

```
[.../exercise]$ ./e4
Even numbers between 0 and 40:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
Numbers 1 to 100:
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
Give a number: 2
The square of 2 is 4
Give a number: 5
The square of 5 is 25
Give a number: 9
The square of 9 is 81
Give a number: 0
You entered zero.
[.../exercise]$
```

Exercise 5 Loops II

a) Write a program that asks for a number. Then the program should print 1 through the given number on separate lines.

b) Encapsulate your code in a while-loop that asks the user if he/she would like to run the program again. Note that when reading a character from the input stream, the newline from the previous input is still buffered and considered as input. To discard the newline, start the *scanf* string with a space like this: `scanf(" %c", &input);`.

The output has to be as follows:

```
[.../exercise]$ ./e5
Give a number: 5
1
2
3
4
5
Run again (y/n)? y
Give a number: 2
1
2
Run again (y/n)? n
Exiting...
[.../exercise]$
```

Exercise 6 Functions

Functions are a great way to make code reusable, improve the structure of the code and isolate errors. Write functions that:

a) take two floats as argument and returns the minimum of those.

b) take four floats as argument and returns the minimum. Make use of the function defined in a).

c) are the same as in a) and b), but returns the maximum.

d) take four floats as argument and returns their sum.

Write a main function that asks the user for four floats and then outputs the minimum, maximum, their sum and mean value. Use the functions from a) - d) to implement this. The output has to be as follows:

```
[.../exercise]$ ./e6
Give four floats: 10.0 -2.3 13.2 20.4
min: -2.300000
max: 20.400000
sum: 41.299999
mean: 10.325000
[.../exercise]$
```

Exercise 7 Functions II

a) Write functions for the four basic mathematical operations addition, subtraction, multiplication and division. Each function has two numbers as parameters and returns the result. Use integers. You do NOT have to do rounding for the division.

b) Write a program that asks the user for numbers a and b, and then use these numbers as arguments for your functions and print the result on the screen.

The output has to be as follows:

```
[.../exercise]$ ./e7
Give a: 11
Give b: 5
11 + 5 = 16
11 - 5 = 6
11 * 5 = 55
11 / 5 = 2
[.../exercise]$
```

Exercise 8 Arrays

In this exercise we look at some basic operations on arrays. Write a C function that:

a) counts the number of 0's in an integer array. The number of 0's is returned by the function.

b) prints an array of integers. The integers are printed on one line, enclosed in curly brackets and separated by commas.

c) triples the value of all elements in an array of integers.

All functions take two parameters, a pointer to the array of integers and the number of elements in the array.

Write a main function that asks the user to input 10 integers and stores them in an array. Use your other functions to print the initial array, the number of zero-valued elements in the array and the contents of the array when all elements have been tripled. The output has to be as follows:

```
[.../exercise]$ ./e8
Input 10 numbers: 1 2 3 0 -3 -2 -1 0 10 11
Initial array: { 1, 2, 3, 0, -3, -2, -1, 0, 10, 11 }
Number of 0's: 2
Tripled array: { 3, 6, 9, 0, -9, -6, -3, 0, 30, 33 }
[.../exercise]$
```

Exercise 9 Pointers and Strings

In this exercise, you will practice how to program with pointers and strings. Without using any library functions, write a C function

```
void append(char* str1, char* str2) { ... }
```

that takes as argument two strings `str1`, `str2` and appends `str2` to `str1`. After calling `append`, the pointer `str1` is supposed to point to the concatenation of (the original) `str1` and `str2`. The caller of `append` has to make sure that enough memory for the result of concatenation is available at the memory address that `str1` points to.

Example

```
char x[12] = { 'H', 'e', 'l', 'l', 'o', ' ',
               0, 1, 2, 3, 4, 5 };
char *y = "world";
append(x, y);
// now "x" contains the string "Hello world"
```

Your implementation needs to make sure that the output string (pointed to by `str1`) remains a well-formed string. Recall that, by definition, a string in C is an array of characters terminated with zero.

Write a main function that asks the user to input 2 words and stores them in character arrays. Then use your `append` function to append the second word to the first, and output the result. The output has to be as follows:

```
[.../exercise]$ ./e9
Enter first word: Hello
Enter second word: World
Result of append: HelloWorld
[.../exercise]$
```

Bonus question: Is it possible that an invocation of `append` changes the string that `str2` points to? Argue why this is not possible, or give an example program where this happens. In the latter case, make sure that your implementation of `append` behaves in an acceptable manner also in such situations (e.g, your program is not supposed to end up in an infinite loop).

Exercise 10 Malloc and sorting strings

In this exercise we are going to implement an algorithm to alphabedically sort character strings. The algorithm we are using is called *bubble sort*, which is an easy, but inefficient sorting algorithm. The algorithm works as follows:

1 We are given an array of comparable elements.

2 We loop through the array, comparing the elements next to each other in the array. If a pair is in the wrong order, we swap their position.

3 If any position was changed, go back to step 2. If we didn't find a pair in the wrong order, we are done.

Our program is going to operate on an input of unkown size, which means that we have to do memory allocation dynamically using `malloc` and `free` from `stdlib.h`.

When comparing strings we use `strcmp` from the library `string.h`. The function `strcmp` considers the ASCII values of the characters when comparing, which means that numbers are considered smaller than uppercase letters which are smaller than lowercase letters.

Write a program that asks the user for how many strings to input, what the maximum string length is and then the actual strings. The program should then output the same strings in alphabetical order (according to `strcmp`). The program should be able to handle an arbitrary number of strings of an arbitrary maximum length. Make sure to free up your allocated memory.

The output has to be as follows:

```
[.../exercise]$ ./e10
Number of strings: 8
Maximum string length: 10
Give string 0: Hello
Give string 1: world!
Give string 2: Here
Give string 3: is
Give string 4: a
Give string 5: big
Give string 6: number:
```

```
Give string 7: 1234567890
Input when sorted:
1234567890
Hello
Here
a
big
is
number:
world!
[.../exercise]$
```

Exercise 11 Sorting arrays in linear time

Write the function threeColorsSort that takes as input an array of integers in the range of 0 and 2 (0, 1 and 2 only), and arranges them in an increasing order:

```
void threeColorsSort(int * theArray, int arraySize)
```

Your solution should have linear runtime in the parameter `arraySize`.

Then, write a program that asks the user for how many numbers to input, and then for the actual numbers. The program should then output the same numbers in ascending order. Make sure to free up your allocated memory.

The output has to be as follows:

```
[.../exercise]$ ./e11
Number of inputs: 5
Give number 0: 2
Give number 1: 1
Give number 2: 0
Give number 3: 1
Give number 4: 1
Input when sorted:
0
1
1
1
2
[.../exercise]$
```

Exercise 12 Recursion

The Fibonacci sequence is a sequence of numbers where the first two numbers are 1 and 1 and the next number in the sequence is the sum of the

previous two numbers. The n'th number in the sequence can be calculated as:
$f(1) = 1$
$f(2) = 1$
$f(n) = f(n-1) + f(n-2)$
See the example for the seven first numbers in the sequence.

a) Write a C function with a parameter n that returns the n'th Fibonacci number. The function must be recursive, i.e., it should call itself.

b) Write a program that asks the user for a number n and then prints the n first numbers in the Fibonacci sequence.

The output has to be as follows:

```
[.../exercise]$ ./e12
Give n: 7
1
1
2
3
5
8
13
[.../exercise]$
```

Exercise 13 Efficient Fibonacci numbers

Write a second function for computing the Fibonacci sequence that uses a loop instead of recursion, and that has linear runtime in the given input. The output has to be as follows (negative numbers occur as a result of arithmetic overflow):

```
[.../exercise]$ ./e13
Give n: 100
1
1
2
3
5
8
13
[...]
708252800
-798870975
-90618175
```

```
-889489150
-980107325
[.../exercise]$
```

Exercise 14 Command line arguments and file I/0

Write a program that outputs a multiplication table. The program takes 2 optional (for the user, not for you) arguments: input file and output file. They are specified with `-in <filename>` and `-out <filename>`. The order should not matter. If no input file is specified, `stdio` is used as input. If no output file is `stdout` is used for output. The user specifies number of rows and columns, in that order, for the multiplication table with two integers. The columns must be aligned for all values not exceeding 1000. Remember to close any files that you opened.

The output has to be as follows:

```
[.../assignment3]$ ./a3e1
4 5
   1    2    3    4    5
   2    4    6    8   10
   3    6    9   12   15
   4    8   12   16   20
[.../assignment3]$ ./a3e1 -in input.txt
   1    2    3    4    5    6
   2    4    6    8   10   12
   3    6    9   12   15   18
   4    8   12   16   20   24
   5   10   15   20   25   30
   6   12   18   24   30   36
   7   14   21   28   35   42
   8   16   24   32   40   48
   9   18   27   36   45   54
  10   20   30   40   50   60
  11   22   33   44   55   66
  12   24   36   48   60   72
[.../assignment3]$ cat input.txt
12 6
[.../assignment3]$ ./a3e1 -out table1.txt
2 3
[.../assignment3]$ cat table1.txt
   1    2    3
   2    4    6
[.../assignment3]$ ./a3e1 -out table2.txt -in input.txt
[.../assignment3]$ cat table2.txt
   1    2    3    4    5    6
```

```
   2    4    6    8   10   12
   3    6    9   12   15   18
   4    8   12   16   20   24
   5   10   15   20   25   30
   6   12   18   24   30   36
   7   14   21   28   35   42
   8   16   24   32   40   48
   9   18   27   36   45   54
  10   20   30   40   50   60
  11   22   33   44   55   66
  12   24   36   48   60   72
[.../assignment3]$
```

Exercise 15 Doubly linked list

Write a program where you declare a structure for a doubly linked list. The struct should contain a pointer to a string (name) as a key, a pointer to the previous element, a pointer to the next element and a birthdate. Please refer to the linked list C-code on the lecture slides for inspiration. Instantiate a list with input from `stdio`. End with inputing `Q` as name. Then output the list sorted alphabetically by name according to ASCII. Your solution should work for an arbitrary number of elements, with a maximum name length of at least 20 characters. Don't forget to free everything you allocate.

The output has to be as follows:

```
[.../exercise]$ ./e15
Name (Q to quit): Felix
Birthdate: 20121224
Name (Q to quit): Erica
Birthdate: 19980613
Name (Q to quit): Dawn
Birthdate: 19831004
Name (Q to quit): Charles
Birthdate: 19670225
Name (Q to quit): Benny
Birthdate: 20010810
Name (Q to quit): Astrid
Birthdate: 19901105
Name (Q to quit): Greg
Birthdate: 19940423
Name (Q to quit): Q
Astrid, 19901105
Benny, 20010810
Charles, 19670225
```

```
Dawn, 19831004
Erica, 19980613
Felix, 20121224
Greg, 19940423
[.../exercise]$
```

Exercise 16 Function pointers

The general understanding of a pointer is the memory address of some kind of data (integer, array, string, structure, etc ...).
A pointer can be as well pointing to a function.

Example Imagine we need to perform several kind of arithmetic operations on integers, let say: multiplying by two, resetting to zero, inverting the integer sign, etc ...
Therefore we define the following functions

```
void op_double(int * a) {...}
void op_reset(int * a) {...}
void op_invert(int * a) {...}
```

Example of use of one of those functions:

```
int a;
a = 5;
op_double(&a);
/* now (a == 10) holds */
```

We can now define a general function pattern using function pointer definition:

```
void (* arithmeticFuncPtr) (int *);
```

Notice that the star is related to the function, not to the returned value. If we consider an integer returning function, we would have the following:

```
int (* funcPtr) (int );
/* funcPtr is a pointer to a function that takes as argument
    an integer an returns an integer.*/

(int *) funcPtr (int );
/* funcPtr is a function that takes as argument
    an integer an returns a pointer to an integer.*/

(int *) (* funcPtr) (int );
/* funcPtr is a pointer to a function that takes as argument
    an integer an returns a pointer to an integer.*/
```

The following code illustrates use of the previously defined arithmetic pointer:

```
int a = 5;

// Assign the op_double function address to the
// function pointer.
arithmeticFuncPtr = &op_double;

// it also works to say: arithmeticFuncPtr = op_double

// Call of the op_double function using
// the arithmeticFuncPtr
(*arithmeticFuncPtr)(&a);

// it also works to say: arithmeticFuncPtr(&a)

// now (a == 10) is true
```

Thanks to their power, function pointers are used often in C libraries, frameworks, and APIs.

Here is what you need to do in this assignment:

1. Write the arithmetic functions `op_double`, `op_reset` and `op_invert`.
2. Write the function `applyTo` that takes as input a function pointer `func`, an array of integers `tab` and the array size `size`, and applies the pointed function to all the elements of the array:

   ```
   void applyTo(void (* func)(int *), int * tab, int size)
   ```

3. Write a main function that asks to user to input a number of integer values, stores the values in an array, and then applies the functions `op_double`, `op_reset` and `op_invert` to the array elements, with the help of `applyTo`.

The output has to look as follows:

```
[.../exercise]$ ./e16
Number of inputs: 5
Give number 0: 10
Give number 1: 9
Give number 2: 8
Give number 3: -1
```

```
Give number 4: -3
Result of applying op_double: { 20, 18, 16, -2, -6 }
Result of applying op_reset: { 0, 0, 0, 0, 0 }
Result of applying op_invert: { -10, -9, -8, 1, 3 }
[.../exercise]$
```

Exercise 17 Hamming weight of integers

Write a program that reads an integer n as input, and outputs the number of bits that are set in the binary representation of n. Can you write this program without using bit-shifts `<<`, `>>` (or multiplication/division by 2)?

The output has to look as follows:

```
[.../exercise]$ ./e17
Enter a number: 42
The number of bits set in 42 is 3
[.../exercise]$
```

Exercise 18 Strongly connected components

A finite directed graph is a tuple (V, E), where V is a finite set of nodes, and $E \subseteq V^2$ a set of directed edges. A strongly connected component (SCC) in a directed graph is a maximum subset $C \subseteq V$ such that the graph contains a path $s_0 \to s_1 \to \cdots \to s_n$ for any two nodes $s_0, s_n \in C$. Efficient computation of SCCs is important in various application domains, and can be used to reduce any graph to a directed acyclic graph (DAG).

Write a program that lets the user input a directed graph and computes the SCCs of the graph; the SCCs have to be output sorted and in lexicographic order. Can you write a program that runs in time linear in the size of the graph?

The output has to look as follows:

```
[.../exercise]$ ./e18
Enter the number of nodes: 4
Graph contains nodes 0, 1, 2, 3
Enter the edges (-1 to terminate):
0 -> 1
1 -> 2
2 -> 1
2 -> 3
-1
The strongly connected components are:
{ 0 }
{ 1, 2 }
{ 3 }
[.../exercise]$
```