##### Appendix

**Exercises**

**Instructions:**

# ***Some instructions about how to do your homework in order to get nice grades and a lot of happiness !***

* Homework will be collected at the beginning of the lesson.
* Do not use // for remarks (although MSDEV understands it), use only /\* \*/.
* Remarks on EVERYTHING, including an explanation at the beginning of the program.
* Indentation (a VERY readable program).
* Meaningful names.
* Efficiency :
  + No repetition of code
  + No repetition of same calculation (for example : sqrt(5) => save it in a variable).
  + No x += 1 nor j = j + 8, use the shortest way to calculate.
* Modularity:
  + Use ‘a’ and not 97.
  + Use #define instead of a variable when data is known (not entered by the user) and should be easily changed.
* Use only standard libraries (no <conio.h> etc.)
* Make sure that it runs correctly (including end cases) before handing it !
* Printed nicely.
* A floppy with an exe.

**Exercise 1 (after end of chapter 3)**

1. Here is a program calculating the sum of two integers.

The program has a number of errors.

Key in the program while correcting the errors.

#**include <stdio.h>**

**main()**

**{**

**short int num1,num2, sum;**

**printf("Enter two integers (-32768 to 32767 ): ");**

**scanf("%f %f", &num1, &num2);**

**sum = num1 + num2;**

**printf("The sum of %d and %d is: %d \n", num1, num2, sum);**

**}**

1. Solve exercise 3.38 (Operators and Expressions – 38), question 1.
2. The aim of the following exercises is to teach and absorb a central idea in programming – dry running.

This refers to “running” code “dryly” (using a follow-up table) and seeing the results.

This is the way prehistoric debuggers worked, but the method is still important today, as this is the way that one learns to read someone else’s code. It is very important **not** to run the code by computer but manually!

* What is the output of the program shown on 3.40, with the variables in line 10 being assigned the value2 and those in line 11 the value 13?

**Exercise 2 (after end of chapter 4)**

1. Solve the exercise in chapter 4, control flow - 37, exercise #1.
2. Solve the exercise in chapter 4, control flow – 38, exercise #5.  
   Do not use the “if” statement.   
   The input must be checked and if it is erroneous (not an alphabetical character) the program is to print an appropriate message.
3. Write a program that identify an input of a character as: alfa (a, G, z, F etc.), number (0-9), space (‘ ‘) or other (!, $, \*, etc.)

**Exercise 3 (after end of chapter 5)**

1. A Fibonacci sequence is a sequence of numbers defined by:   
   fib(n) = fib(n-1) + fib(n-2), i.e. each number in the sequence is equal to the two preceding numbers (1 1 2 3 5 8 …) apart from fib(1) and fib(2) that are equal to one another (there is no fib(0)).  
   The sequence can also be calculated using the following formula:  
     
     
     
     
   Write a program calculating the first 20 numbers of the sequence by the formula.



Write the program in he most efficient way possible **without** using the function **pow()**.

The output is to be in colums as follows, and the title is to be:  
 n f(n) f(n) rounded

1. What are the values of variables a, b, c for each of iteration of the following program? Use a table.

**#include <stdio.h>**

**void main()**

**{**

**int a = 1, b = 1, c, i;**

**for(i = 0; i < 10; i++)**

**{**

**c = b + a;**

**a = ++b;**

**b -= c--;**

**}**

**}**  
  
  
  
  
**Exercise 4 (after end of chapter 7)**

1. Write a function called **same\_chars()** with the following appearance:  
   **int same\_chars(char s1[], char s2[], char different[])**The function checks to see whether in the two strings s1 and s2 there are the same characters or not.

If so, the function returns 1 and shall not relate to the variable ‘different’,

Otherwise 0 is returned and the variable ‘different‘(an array sized one) will be the first different character found.  
A sample run of the program is to look like this:  
  
Enter first string: sgggsffds  
Enter second string: gfdshsdfghdfg  
The strings are NOT composed of the same characters.  
’h’ is found in one but not the other.  
  
Enter first string: qywterytrewqqqr  
Enter second string: rewqyteey  
The strings are composed of the same characters.

1. The following lines define three variables.

For each, state:  
When is it allocated?

* 1. Is it initiated?
  2. When is the memory released?

**int i1;**

**void foo()**

**{**

**int i2;**

**static int i3;**

**...**

**}**

**Exercise 5 (after end of chapter 8)**

1. Write the function:  
     
   **int GlobalReplace(char \*string, char \*oldStr, char \*newStr)**  
     
   that replaces all occurrences of the substring ‘oldStr’ in ‘string’ with ‘newStr’.

The function returns the number of replacements made, or zero, if none were made.

Use pointers rather than indexes.  
The function must also work in the case of ‘newStr’ being longer than ‘oldStr’ and when ‘newStr’ is an empty string (“”). – check validity!

**Exercise 6 (after end of chapter 9)**

1. Write a program that calls a number of strings of varying length not known in advance.

The number of strings being called form the user at the beginning. After reading the strings, they are to be inserted into an array of pointers to be transferred to the function **concat().**

Write this function too.  
  
**char\* concat(int numOfStrings,char\* strings[],int totalLen)**   
  
The function receives:

* + An array of strings.
  + A variable stating the number of strings transferred.
  + The total length of all the strings.

The function returns one long string containing the concatenation of all strings one after the other.

Handle the case in which the string array transferred is empty(NULL). Do not use a string of known size at any point in the function, but dynamic allocations only!  
Assume that the string array is continuous without any empty strings.

**Exercise 7 (after end of chapter 10)**

1. Write a program that inputs the student’s details (name and marks) and calculates his average mark in each course and average mark for the semester.  
     
   A student’s semester mark is calculated as follows:  
   * 1/3 of his mark is the average of his marks in three home exercises.
   * 1/3 of his mark is the average of his marks in two mid-semester exams.
   * 1/3 of his mark is the final exam mark.

A student studies two courses:

* + C programming and data structures.
  + Use the following two structures:
    - structure STUDENT and structure COURSE.
  + Write the program modularly, so that the various actions, such as inputting student details, calculating average mark for each course, etc., are performed in separate functions.
  + Resolve the functions so that the number of courses/ exams/ exercises may be changed without modifying the code.
  + Illogical input (a mark in excess of 100 or below 0) will generate an appropriate error and halt the program.
  + Make sure all names are meaningful.
  + Provide a short explanation at the beginning of each function concerning its action.
  + Wherever the code’s purpose is not obvious, provide an explanatory comment.

An example of a run of this program:

Enter your name: Michelle

Data structures:

==========

Enter your grade in EX1: 90

Enter your grade in EX2: 100

Enter your grade in EX3: 90

Enter your first mid-term grade: 80

Enter your second mid-term grade: 72

Enter your final mid-term grade: 83

# C Programming

=========

Enter your grade in EX1: 70

Enter your grade in EX2: 95

Enter your grade in EX3: 80

Enter your first mid-term grade: 80

Enter your second mid-term grade: 87

Enter your final mid-term grade: 89

Michelle, your final grades are:

Data structures: 84.11

C Programming: 84.72

The semester grade: 84.41

1. Dynamic memory allocation is used when the programmer does not know, in advance, how much memory will be needed. In such a case, to store a list of elements (integers, structures, etc.), a **linked list** is often used instead of a static array.

A linked list is a list of **nodes**, each node containing **data** and a **pointer** to the next node in the list. The last node’s pointer is NULL and the programmer has a pointer to the first node in the list.



The programmer dynamically allocates/deallocates space as needed, adding to or deleting from the linked list and performing other tasks.

**/\* this is the node for the linked list \*/**

**typedef struct node {**

**int data;**

**struct node \*next; /\* each structure contains a pointer to**

**a structure like itself \*/**

**} NODE, \*PNODE;**

**PNODE first; /\* pointer to first node in list \*/**

What does the function you\_tell\_me() do? It is important that you answer the question yourself!

Hint: Use the linked list that you see on this page as an example. The numbers that you see are the data fields.



**void you\_tell\_me()**

**{**

**PNODE origfirst, next, current;**

**if(first == (PNODE)NULL) /\* if the list is empty \*/**

**{**

**printf("empty list\n");**

**return;**

**}**

**/\* if the list is not empty \*/**

**origfirst = first;**

**current = first;**

**next = current->next;**

**while(next)**

**{**

**first = next;**

**next = first->next;**

**first->next = current;**

**current = first;**

**}**

**origfirst->next = NULL;**

**}**