Programming in C

Table of Contents

*1* *Introduction* 2

*2* *Part A* 2

2.1 Prerequisites 2

2.2 Change value via a pointer 2

2.3 Metric to imperial conversion 2

2.4 Value swap 3

*3* *Part B* 4

3.1 Prerequisites 4

3.2 Unit testing 4

# Introduction

This document describes the assignments for the first weeks of the course Programming in C

For the first assignments, create your source code from scratch and compile on the command line.

# Part A

## Prerequisites

Please study the modules about pointers.

## Change value via a pointer

Write a program to define one integer and two pointers to integer. Both pointers point to the same integer. Change the integer value via one pointer and read it back via the other pointer.

Int I = 5;

Int\* pointer1 = NULL;

Int\* pointer2 = NULL;

pointer1 = &I;

pointer2 = &I;

\*pointer1 = 4;

printf(“Waarde:%d”,\*pointer2);

## Metric to imperial conversion

When you want to return a value from a function, you can simply return that value. What happens when you need to return more than one value? In that case you can use a pointer type parameter. Please implement the following function that can convert your body length in centimetres to your length in feet and remaining inches:

void metricToImperial(int lengthInCm,

int\* feet,

int\* remainingInches)

|  |
| --- |
| 1 Foot = 30.48 cm |
| 1 Inch = 2.54 cm |

A conversion of 176 centimetres should result in:

**5** foot 🡨 176/ 30.48

and

9 inches 🡨 (176- **5**\*30.48 ) / 2.54

void metricToImperial(int lengthInCm,

int\* feet,

int\* remainingInches)

{

\*feet = lenghtInCm / 30.48;

lenghtInCm = lengthInCm – 5\*30.48;  
 \*remainingInches = lenghtInCm / 2.54;

}

void main(){

int length = 176;

int feet;

int inches;

metricToImperial(length,&feet, &inches);

printf(“Feet:%d”,feet);

printf(“Inhes:%d”, inhes);}

## Value swap

Write a function that can swap two integer values. The function is called swapValues(), the following code snippet explains what it does:

int value1 = 35;

int value2 = -97;

swapValues(&value1, &value2);

// now value1 equals -97 and value2 equals 35.

The following drawing explains this in a bit more detail. The italic numbers below the boxes represent fictive addresses; the boxes on the left are the parameters of swapValues():

SwapValues(\*val1, \*val2) {

Int tempval1;

Int tempval2;

Tempval1 = \*val1;

Tempval2 = \*val2;

\*val1 = tempval2;

\*val2 = tempval1;

}

-97

value1

Address:3000

35

value2

Address: 3004

SwapValues (&value1, &value2);

35

value1

Address:3000

-97

value2

Address: 3004

# Part B

## Prerequisites

Please study the modules about unit testing.

## Unit testing

Make unit tests for module buffer in the demo\_UnitTests project directory (found in StartPointForExercises). The goal of this exercise is to find the bugs in the buffer implementation by creating a complete set of unit tests and using your pointer and array skills.

Please read through the c- and h-files in the project directory.

The file buffer.c implements a buffer of 20 bytes (uint8\_t):

* unsigned 8-bit values (uint8\_t) can be read and written.
* unsigned 32-bit values (uint32\_t) can be read and written.

**Example**

Step-1: The situation where the buffer only contains zero’s by calling: initBuffer();

Step-2: The situation after calling: writeBufferValueAsUint8(3, 32);

Step-3: The situation after calling: writeBufferValueAsUint32(1, 2155905152);

The number 2155905152 is the binary number: 1000 0000 1000 0000 1000 0000 1000 0000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Byte-1 | Byte-2 | Byte-3 | Byte-4 |
| Bit-value | 1000 0000 | 1000 0000 | 1000 0000 | 1000 0000 |
| Decimal-value | 128 ( = 27) | 128 | 128 | 128 |

Check for yourself via [calculator.net](http://www.calculator.net/binary-calculator.html?b2dnumber1=10000000100000001000000010000000&calctype=b2d&x=64&y=20#binary2decimal). This number is stored in a 32-bit / 4 byte variable as:

State of the buffer after the different steps is given in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Buffer Values after step-1 | Buffer Values after step-2 | Buffer Values after step-3 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | **128** |
| 2 | 0 | 0 | **128** |
| **3** | **0** | **32** | **128** |
| 4 | 0 | 0 | **128** |
| 5 | 0 | 0 | 0 |
| … |  |  |  |
| 16 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 |
| 18 | 0 | 0 | 0 |
| 19 | 0 | 0 | 0 |

**Assignment**

Repeat the following steps until all methods as given in buffer.h are tested sufficiently:

* Add a test case to ‘buffer\_test.c’
* Run the new set of test-cases (‘make test’)
* When needed extend and/or fix the code in ‘buffer.c’ to PASS all testcase(s).

# 