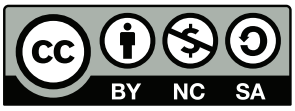
Learn programming by example Jos

Advanced C programming

a collection of topics for experienced C programmers

August 2020

Jos Onokiewicz



Foreword

You should have a thorough basic level of C programming experience to make the most of this book about advanced C programming. I therefore assume you are already able to use different data types, arrays, structs, pointers, loops, selections, functions, typedefs, define constants, arithmetic and logical expressions. There are many good comprehensive C programming books, videos and tutorials abundantly available. Initially this book will help you avoid the great quest for reliable sources. I have added several references in the bibliography for the purpose of consulting. This bibliography can be used as a starting point for consulting other more profound sources about C programming and software engineering.

The programming topics are described in self-contained chapters. Each chapter contains one or more exercises. Each topic is accompanied by a least one full compilable small code example. Just by thoroughly reading the code examples it should increase your level of understanding the C language. It requires no further explanation that compiling, running and doing coding experiments plays a part in improving your programming skills. All examples are programmed in the same coding style and use the same naming conventions. I have strived to achieve a high level of readability. All the code sources are available on GitLab [12]. The code examples must be compiled by a C99 compiler.

In each chapter I have added a quotation of software engineering wisdom. Perhaps, at first sight, you will not fully understand them, but I urge you to reread them during your programming exercises and hopefully, with each time, you will better understand them and find them to be meaningful.

Thanks to Ghedhban Swadi, Hugo Arends, Ton Ammerlaan and Remko Welling for their supportive comments in order to improve this book.

LaTeX was used for the preparation of this text.

This book (version 2.1) is still under construction. Comments, therefore, are greatly appreciated.

Jos Onokiewicz

List of abbreviations

ADT	Abstract Data Type Model for data types where a data type is described by its behaviour from the point of view of a user of the data. A data type is a set of values and a collection of operations on those values.
ANSI	American National Standards Institute
ANSI C	ANSI C standard Also known as C89 (or C90 essentially the same language)
API	Application Programmers Interface
DLL	Doubly Linked List
C11	C standard 2011
C99	C standard 1999
CLI	Command Line Interface
CSLL	Circular Singly Linked List
DUT	Device Under Test
FIFO	First In First Out, queue
IDE	Integrated Development Environment For example: QtCreator, CLion, Visual Studio Code and Eclipse.
ISO	International Organization for Standardization
LOC	Lines of code
LIFO	Last In First Out, stack
OS	Operating System
SDLC	Software Development Life Cycle Requirement gathering, system analysis, design, coding, testing, integration, maintenance and documentation
SLL	Singly Linked List
stderr	standard error file stream
stdin	standard in file stream
stdout	standard out file stream
TDD	Test Driven Development
UUT	Unit Under Test

Contents

It's OK to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it.

Steve McConnell

1

Pointers, arrays and functions

1.1 Introduction to pointers

A pointer represents a memory location. Every memory location (memory address) has its own unique identifier (unsigned integer value). Memory can be thought of simply as an array of bytes. Every variable and function in your program is located at some unique memory address. Because variables are typed, pointers also need to be typed to be able to dereference a pointer and to perform pointer arithmetic on.

The size of a pointer value and the size of what it points to are not related.

Four arithmetic operators can be used on pointers: +, −, ++ and −−. Pointer arithmetic uses always the number of bytes of its data type. So incrementing an integer pointer by 1 will add the number of bytes equal to the size of an *int* type. Generally pointer arithmetic if used for traversing through an array. A C compiler does not add code for checking if a pointer is out of array bounds.

You can not add or multiply two pointer values. This will result in a compiler error, because these operations are meaningless. You can add positive and negative integer values (offsets) to pointers. It is possible to subtract two pointer values for calculating the distance in memory expressed in a multiple of the size of the data type. A pointer difference can be typed as *ptrdiff_t* defined in *stddef.h*. For avoiding undefined behaviour these pointers must point to elements of the same array or just one past the last element of the array. Pointers pointing to elements of the same array can be compared by using relational operators, such as ==, < and >. Because array elements are guaranteed contiguous in memory.

You can obtain the address of a variable by using the address operator & preceding the variable name. You can dereference a pointer for accessing the data that the pointer is pointing to, by using the dereference operator * prefixing the pointer. Dereferencing an *int* pointer will result in an integer value. Dereferencing a *double* pointer will result in a *double* value.

A pointer variable contains a memory address. Because pointer variables are typed the compiler can check to which kind of data type the pointer is pointing. Naming convention for pointer variable names: prefix variable names with a lower case *p*. Use an uppercase character after the *p* (*lowerCamelCase* naming convention). This will improve the readability and understandability of programs. The next code example shows the defining declaration and initialisation syntax of several different typed pointer variables .

Listing 1.1: pointerVariables/main.c

```
1  #include <stddef.h>
2  #include <stdio.h>
3
4  #define DATASIZE 5
5
6  int main(void)
7  {
8     int iVar = 1;
9     double dVar = 2.2;
10    char cVar = 'A';
11    int data[DATASIZE] = {1, 2, 3, 4, 5};
12
13    /* Pointer variables of different types */
14    int *pI = &iVar;
15    double *pD = &dVar;
16    char *pC = &cVar;
17    int *pData = data;
18
19    printf(" Size pointer variable pI = %lu bytes\n", sizeof(pI));
20    printf(" Size pointer variable pD = %lu bytes\n", sizeof(pD));
21    printf(" Size pointer variable pC = %lu bytes\n\n", sizeof(pC));
22
23    printf(" pI %p points to %d\n", pI, *pI);
24    printf(" pD %p points to %lf\n", pD, *pD);
25    printf(" pC %p points to '%c'\n\n", pC, *pC);
26
27    /* pData == &data[0] */
28    printf(" pData = data;  pData %p points to %d\n", pData, *pData);
29
30    /* pData is a pointer variable, so you can update the value, for
31     * instance pData++.
32     * data++ is not possible, this will result in a compiler error,
33     * because you can not assign a new pointer value to an array variable.
34     * An array variable is memory bound.
35     */
36    pData++;
37    /* pData == &data[1] */
38    printf(" pData++;       pData %p points to %d\n", pData, *pData);
```

```
39
40     pData += 3;
41     /* pData == &data[4] */
42     printf(" pData += 3;    pData %p points to %d\n\n", pData, *pData);
43
44     /* A pointer difference is typed as ptrdiff_t in stddef.h */
45     ptrdiff_t pdif = pData - data;
46     printf(" pData points to data[%ld] = %d\n\n", pdif, data[pdif]);
47
48     pData += 10;
49     /* Array bounds checking */
50     if ((pData < data) || (pData > data + DATASIZE))
51     {
52        fprintf(stderr, " ERROR: pData accessing 'data' out of bounds\n\n");
53     }
54
55     return 0;
56 }
```

Always use a preprocessor constant macro to represent the size of arrays. This will improve readability and maintainability of the code.

The output shows the pointer values in hexadecimal format and the values they point to. All pointer variables are 8 bytes sized. Rerunning this program will result in other values for these pointers:

```
Size pointer variable pI = 8 bytes
Size pointer variable pD = 8 bytes
Size pointer variable pC = 8 bytes

pI 0x7ffd854000bc points to 1
pD 0x7ffd854000c0 points to 2.200000
pC 0x7ffd854000bb points to 'A'

pData = data;  pData 0x7ffd854000f0 points to 1
pData++;       pData 0x7ffd854000f4 points to 2
pData += 3;    pData 0x7ffd85400100 points to 5

pData points to data[4] = 5

ERROR: pData accessing 'data' out of bounds
```

Pointer values can be assigned to other pointer variables of the same type.

We can not assign the address of an *int* to a *double* pointer variable without explicit type casting, see paragraph 1.5 for details about type casting of pointers.

1.2 NULL pointers

A *NULL pointer* is a defined value for pointing to 'nothing'. This is a well-defined memory location (in most compiler implementations this is a zero-valued integer). *NULL* is a preprocessor constant macro defined in *stddef.h*.

It is good practice to assign this *NULL* value to pointers at the moment of declaration if another defined value can not be assigned. Avoid uninitialised pointers because an uninitialised pointer can point anywhere in memory. Dereferencing an uninitialised pointer will result in undefined behaviour of your application.

Never use a dereferenced *NULL* pointer for assigning a value as this causes undefined behaviour of the application. Because you are copying this assigned value in the memory address *NULL*. This address is often used for system purposes, so this can cause a system crash.

1.3 Arrays, array index, array size and pointers

The variable name of an array (fixed sized data structure) will be converted to a pointer to the first element in the array if used as a function parameter. An array can not be passed by value instead a pointer to the array is passed by value. Passed by value means that a copy of the actual parameter value is passed (see paragraph 1.6 for more about functions using pointer parameters).

You can not assign a new pointer value to the name of an array variable. Copying an array to another array cannot be done by an assignment. This will result in a compiler error. Copying *char* arrays (null character terminated strings) can be done by the function *strcpy()* or *strncpy()*, other array types can be copied by *memcpy()*. Or you use a loop for copying every single data member.

The next code example calculates the number of data values in an array and shows that you can not calculate the number of data values in an array passed by pointer in a function. If you pass an array to a function you should add the size of the array as an additional function parameter. According to the C99 standard *size_t* is an unsigned integer type of at least 16 bit. This type guarantees to hold any array index value. The implementation is compiler dependant, for example:

```
typedef long unsigned int size_t;
```

For printing *size_t* typed values C99 uses the %*zu* format specifier, in C89 you should have used %*lu*.

The next program will show some compiler warnings: *'sizeof' on array function parameter 'b' will return size of 'const double *'* .

Listing 1.2: arraySize/main.c

```
1  #include <stdio.h>
2
3  #define ARRAYSIZE 4
4
5  void printArraySize(const double b[]);
6
7  int main(void)
8  {
9     double buffer[ARRAYSIZE] = {1.0, 2.0, 3.0, 4.0};
10
11    printf("Size double    = %zu bytes\n\n", sizeof(double));
12    /* buffer is an array type variable */
13    printf("Size buffer    = %zu bytes\n", sizeof(buffer));
14    printf("Size buffer[0] = %zu bytes\n", sizeof(buffer[0]));
15
16    size_t size = sizeof(buffer) / sizeof(buffer[0]);
17    printf("Number of data values in buffer = %zu  ", size);
18
19    if (size == ARRAYSIZE)
20    {
21       printf("is correct\n");
22    }
23    puts("");
24
25    /* buffer becomes pointer type! */
26    printArraySize(buffer);
27
28    return 0;
29 }
30
31 void printArraySize(const double b[])
32 {
33    /* b is an array pointer type: double* */
34    printf("Size b         = %zu bytes\n", sizeof(b));
35    printf("Size b[0]      = %zu bytes\n", sizeof(b[0]));
36
37    size_t size = sizeof(b) / sizeof(b[0]);
38    printf("Number of data values in array b = %zu  ", size);
39
40    if (size != ARRAYSIZE)
41    {
42       printf("is not correct\n");
43    }
44 }
```

The output of the previous code example shows the results of calculating the number of data elements in an array, using the *sizeof()* function:

```
Size double    = 8 bytes

Size buffer    = 32 bytes
Size buffer[0] = 8 bytes
Number of data values in buffer = 4  is correct

Size b         = 8 bytes
Size b[0]      = 8 bytes
Number of data values in array b = 1  is not correct
```

If you use a function with a pointer to an array, you must also add to the parameter list an input variable containing the size of the array for traversing the array in a local for-loop in the function.

Arrays can also contain pointers. For example *stringData* is an array holding *char* pointers for pointing to various strings. This array is typed as *char**. The characters in the strings are not contained in the array *stringData*. The compiler has put these string literals somewhere else in memory.

Listing 1.3: pointerArray/main.c

```
1  #include <stdio.h>
2
3  #define SIZE 5
4
5  int main(void)
6  {
7     /* Every string should be considered a pointer to the string */
8     const char *stringData[SIZE] = {"Hello", "world", "of", "pointers",
9                                     "in arrays"};
10
11    for (int i = 0; i < SIZE; i++)
12    {
13       printf("stringData[%d] = %p  points to \"%s\"\n", i, stringData[i],
14              stringData[i]);
15    }
16    puts("\n");
17
18    /* Only printing the first character of every string */
19    puts("First character of every string:");
20    for (int i = 0; i < SIZE; i++)
21    {
22       printf("stringData[%d][0] = '%c'\n", i, stringData[i][0]);
23    }
24
```

```
25     return 0;
26 }
```

Output:

```
stringData[0] = 0x55ffde45f858  points to "Hello"
stringData[1] = 0x55ffde45f85e  points to "world"
stringData[2] = 0x55ffde45f864  points to "of"
stringData[3] = 0x55ffde45f867  points to "pointers"
stringData[4] = 0x55ffde45f870  points to "in arrays"


First character of every string:
stringData[0][0] = 'H'
stringData[1][0] = 'w'
stringData[2][0] = 'o'
stringData[3][0] = 'p'
stringData[4][0] = 'i'
```

The array variable *stringData* points to its first element a *char** type. So *stringData* is typed as a pointer to a pointer: *char* $**$. The previous program shows the notation for a 2-dimensional array: *stringData*[*i*][0].

See the next paragraph for 1-, 2- and 3-dimensional arrays.

1.4 Multi-dimensional arrays

In C you can create multi-dimensional arrays. A two-dimensional array looks like an array of arrays. A three-dimensional arrays is an array of arrays of arrays. You can have arrays with any number N of dimensions. N-dimensional arrays are laid out contiguously in memory. A two dimensional array (also known as a *matrix*, a table of rows and columns) is allocated in row-major order: all the values in row 0 first, followed by all the values in row 1, etc.

Passing N-dimensional arrays to functions, you need to add a size parameter for the first dimension and all the other dimensions $(N - 1)$ for the typing of the N-dimensional array. The next code example shows how you can initialise one, two and three dimensional arrays and how you can pass them to a function.

Listing 1.4: multidimArrays/main.c

```
1  #include <stdio.h>
2
3  #define MAX_ROW 4
4  #define MAX_COLUMN 3
5  #define MAX_DEPTH 2
6
7  void print1DimArray(int ar1D[], size_t size);
8  void print2DimArray(int ar2D[][MAX_COLUMN], size_t size);
9  void print3DimArray(int ar3D[][MAX_COLUMN][MAX_DEPTH], size_t size);
10
11 int main(void)
12 {
13    int data1D[MAX_ROW] = {1, 2, 3, 4};
14
15    int data2D[MAX_ROW][MAX_COLUMN] = {
16       {1, 2, 3}, {10, 20, 30}, {100, 200, 300}, {1000, 2000, 3000}};
17
18    int data3D[MAX_ROW][MAX_COLUMN][MAX_DEPTH] = {
19       {{100, 110}, {200, 210}, {300, 310}},
20       {{101, 111}, {201, 211}, {301, 311}},
21       {{102, 122}, {202, 222}, {302, 322}},
22       {{103, 133}, {203, 233}, {303, 333}}};
23
24    printf("---- 1 dimensional array (%d):\n", MAX_ROW);
25    print1DimArray(data1D, MAX_ROW);
26
27    printf("\n---- 2 dimensional array (%d x %d):\n", MAX_ROW, MAX_COLUMN);
28    print2DimArray(data2D, MAX_ROW);
29
30    printf("\n---- 3 dimensional array (%d x %d x %d):\n", MAX_ROW, MAX_COLUMN,
31           MAX_DEPTH);
32    print3DimArray(data3D, MAX_ROW);
33
34    return 0;
35 }
36
37 void print1DimArray(int ar1D[], size_t size)
38 {
39    for (size_t i = 0; i < size; i++)
40    {
41       printf("%4d ", ar1D[i]);
42    }
43    puts("");
44 }
45
46 void print2DimArray(int ar2D[][MAX_COLUMN], size_t size)
47 {
48    for (size_t i = 0; i < size; i++)
49    {
50       for (size_t j = 0; j < MAX_COLUMN; j++)
51       {
52          printf("%4d ", ar2D[i][j]);
53       }
54       puts("");
55    }
56 }
```

```
57
58 void print3DimArray(int ar3D[][MAX_COLUMN][MAX_DEPTH], size_t size)
59 {
60    for (size_t k = 0; k < MAX_DEPTH; k++)
61    {
62       for (size_t i = 0; i < size; i++)
63       {
64          for (size_t j = 0; j < MAX_COLUMN; j++)
65          {
66             printf("%4d ", ar3D[i][j][k]);
67          }
68          puts("");
69       }
70       puts("");
71    }
72 }
```

The output shows the contents of a 1D array:

```
---- 1 dimensional array (4):
   1    2    3    4
```

Output for a 2D array:

```
---- 2 dimensional array (4 x 3):
   1    2    3
  10   20   30
 100  200  300
1000 2000 3000
```

Output for a 3D array:

```
---- 3 dimensional array (4 x 3 x 2):
 100  200  300
 101  201  301
 102  202  302
 103  203  303

 110  210  310
 111  211  311
 122  222  322
 133  233  333
```

1.5 Type casting pointers

Sometimes it is necessary to change a pointer type to another one using an explicit typecast. Type casting of pointers does not change the value of pointers and is done during compilation-time (no run-time overhead).

Many communication protocols are byte oriented. Therefore you must be able to process different types of data by approaching this data by a byte pointer typed as *unsigned char** or as *char**. Every type of contiguous data can be traversed as an array of bytes. You only need a byte pointer pointing to the beginning and the number of bytes.

The next program uses a function *hexdumpInt*() to show all separate bytes of an *int* in hexadecimal format. We need to typecast an *int* pointer to an *unsigned char* pointer (byte pointer). If you increment the latter *unsigned char* pointer this pointer will point to the next byte in the integer.

Negative numbers are represented by their *two's complement* of their absolute value. Example calculating integer value −1 four byte pattern in hexadecimal format:

```
-1 = ?

 1         = 00 00 00 01   absolute value
 flip bits = ff ff ff fe   1 complement
 add 1     = ff ff ff ff   2 complement

-1 = ff ff ff ff
```

Listing 1.5: hexdump/main.c

```
1  #include <stdio.h>
2
3  int isLittleEndian(void);
4  void hexdumpInt(int data);
5
6  int main(void)
7  {
8     int i1 = 0;
9     int i2 = 1;
10    int i3 = 255;
11
12    if (isLittleEndian())
13    {
14       puts("     Little endian integers:\n");
15    }
16    else
17    {
18       puts("     Big endian integers:\n");
19    }
20
21    hexdumpInt(i1);
22    puts("");
23    hexdumpInt(i2);
24    hexdumpInt(-i2);
25    puts("");
26    hexdumpInt(i3);
27    hexdumpInt(-i3);
28
29    return 0;
30 }
31
32 int isLittleEndian(void)
33 {
34    const int test = 1;
35    const unsigned char* pByte = (const unsigned char*)&test;
36
37    return pByte[0] == 1;
38 }
39
40 void hexdumpInt(int data)
41 {
42    const unsigned char *pData = (const unsigned char *)&data;
43
44    printf("%10d = ", data);
45    for (size_t i = 0; i < sizeof(int); i++)
46    {
47       printf("%02x ", pData[i]);
48    }
49    printf("\n");
50 }
```

The output shows that the size of an *int* variable is 4 bytes and these bytes are sequential ordered in *little endian* format (the least significant byte at the first location in memory).

```
Little endian integers:

   0 = 00 00 00 00

   1 = 01 00 00 00
  -1 = ff ff ff ff

 255 = ff 00 00 00
-255 = 01 ff ff ff
```

Exchanging data by a serial link, network connection or binary files between *little endian* and *big endian* (the most significant byte at the first location in memory) systems causes run-time problems because the order of the bytes in the same typed variables differs for the different communicating systems.

Always check if the communication code you have used has solved the endianness problem. If not, you will need to implement the correct endianness conversion yourself.

1.6 Functions using pointer arguments, call by reference

C uses the *call by value* method for passing function arguments. A C-function copies the values of all arguments. Changes made to a function parameter inside the function will have no effect to the outside value. To change a value outside a function you need to use a *call by reference*. In this case you must use a pointer for passing the function argument. When a pointer to the variable you want to change has passed, the pointer itself is copied. But this copy points to the variable outside the function. Now you can change the value of this variable indirectly by this pointer in the function by dereferencing this pointer.

It is very easy to make mistakes in using the $*$ and & operators. Sometimes your code is still compilable but will result in undefined behaviour of your system. Compiler warnings often indicate these kind of run-time problems.

Listing 1.6: swap/main.c

```
1  /* C uses call by value for passing function arguments.
2   * We need a call by reference for changing an argument value
3   * outside the function.
4   */
5
6  #include <stdio.h>
7
8  void swapByValue(int a, int b);
9  void swap(int *pA, int *pB);
10
11 int main(void)
12 {
13    const int init_x = 2;
14    const int init_y = 3;
15    int x = init_x;
16    int y = init_y;
17
18    printf("Initial values      x = %d  y = %d\n\n", x, y);
19    swapByValue(x, y);
20    printf("swapByValue(x, y);  x = %d  y = %d", x, y);
21    if (x == init_y && y == init_x)
22    {
23       puts("  values are swapped");
24    }
25    else
26    {
27       puts("  values are not swapped\n");
28    }
29
30    /* Call by reference: using pointers */
31    swap(&x, &y);
32    printf("swap(&x, &y);       x = %d  y = %d", x, y);
33    if (x == init_y && y == init_x)
34    {
35       puts("   values are swapped");
36    }
37    else
38    {
39       puts("   values are not swapped\n");
40    }
41
42    return 0;
43 }
44
45 /* Call by value */
46 void swapByValue(int a, int b)
47 {
48    int temp = a;
49    a = b;
50    b = temp;
51 }
52
53 /* Call by reference: using pointers */
54 void swap(int *pA, int *pB)
55 {
56    int temp = *pA;
```

```
57      *pA = *pB;
58      *pB = temp;
59  }
```

Output:

```
Initial values      x = 2  y = 3

swapByValue(x, y);  x = 2  y = 3   values are not swapped

swap(&x, &y);       x = 3  y = 2   values are swapped
```

Arrays are always passed by reference (by a pointer). Because a reference points only to the beginning of the array, you always need to add the size of the array as an additional parameter to the function. This is not necessary for ASCII strings because they contain a string terminating null character.

If you do not want an array to be changed in a function, you must add *const* to the type name of the array. The compiler will show an error if an assignment to an element of the array was programmed in the function.

If you want to change the value of a pointer variable by a function, you must use a pointer to a pointer typed argument. For example as mentioned in the next two functions:

```
void addSLL(node_t **ppHead, int data)
```

```
void clearSLL(node_t **ppHead)
```

in the code example in paragraph 6.2. Naming convention: a pointer to a pointer variable name starts with 2 lower case *p*'s.

1.7 Pointers and bits

A pointer can not point to one single bit. The smallest addressable unit in C is a byte. If you want to read or change a bit value in a byte, you need bitwise operators that will do this. The next code example shows the dump of an array of bytes to *stdout* and some bit operations implemented in functions.

Listing 1.7: bitsbytes/main.c

```
1  #include <stdio.h>
2
3  #define NBYTES 10
4
5  typedef unsigned char byte_t;
6
7  void hexDump(const byte_t *pbMem, int size);
8  void setBit(byte_t *pByte, int iBit);
9  void resetBit(byte_t *pByte, int iBit);
10
11 int main(void)
12 {
13    byte_t *pMemory = (byte_t *)0x00030200;
14    byte_t bytesMemory[NBYTES] = {0x00, 0x11, 0x22};
15
16    puts("Bytes in bytesMemory[NBYTES]:");
17    hexDump(bytesMemory, NBYTES);
18    puts("");
19
20    /* Set byte in 0x0003020 to 0xAA, execution is avoided */
21    /* *(pMemory + 0x0B) = 0xAA; */
22
23    puts("Set bit 3 in byte 5 in bytesMemory[NBYTES]:");
24    setBit(bytesMemory + 5, 3);
25    hexDump(bytesMemory, NBYTES);
26    puts("");
27
28    puts("Set bit 5 in byte 6 in bytesMemory[NBYTES]:");
29    setBit(bytesMemory + 6, 5);
30    hexDump(bytesMemory, NBYTES);
31
32    return 0;
33 }
34
35 void hexDump(const byte_t *pbMem, int size)
36 {
37    int offset = 0;
38    while (offset < size)
39    {
40       printf("%02x ", *(pbMem + offset));
41       offset++;
42    }
43    puts("");
44 }
45
46 /* Bit numbering in byte 76543210, 0 <= bit < 8 */
47 void setBit(byte_t *pByte, int iBit)
48 {
49    const byte_t mask = 0x01 << iBit; /* << shift left operator */
50    *pByte |= mask;
51 }
52
53 /* Bit numbering in byte 76543210, 0 <= bit < 8 */
54 void resetBit(byte_t *pByte, int iBit)
55 {
56    const byte_t mask = ~(0x01 << iBit); /* << shift left operator */
```

```
57     *pByte &= mask;
58 }
```

Output:

```
Bytes in bytesMemory[NBYTES]:
00 11 22 00 00 00 00 00 00 00

Set bit 3 in byte 5 in bytesMemory[NBYTES]:
00 11 22 00 00 08 00 00 00 00

Set bit 5 in byte 6 in bytesMemory[NBYTES]:
00 11 22 00 00 08 20 00 00 00
```

1.8 Local static variables in functions

Local non static variables in functions are created on the stack of the system. Never return a pointer value to a local non static variable in a function. Because the local variable becomes unavailable once the function finishes execution. Local non static variables have limited lifetime. The stack will be used for other local non static variable values.

The next code example shows the usage of static local variables in functions. Static local variables are not created on the stack but on the heap (pre-reserved memory area allocated at the start-up of the program). The lifetime of these variables equals the lifetime of the whole program. Local static variables are initialised only once. Local static variables will initialise to 0 if no initialiser is specified. Functions can return a pointer to a static local variable.

Listing 1.8: static/main.c

```
1  #include <stdio.h>
2
3  int f1(void);
4  int f2(void);
5  int addingup(int delta);
6
7  int main(void)
8  {
9     int resetValue = 0;
```

```
10
11     printf("\n f1() called 4 times, returned values:  %d ", f1());
12     printf(" %d ", f1());
13     printf(" %d ", f1());
14     printf(" %d\n", f1());
15
16     printf(" f2() called 4 times, returned values:  %d ", f2());
17     printf(" %d ", f2());
18     printf(" %d ", f2());
19     printf(" %d\n\n", f2());
20
21     printf(" addingup(0)  = %d\n", addingup(0));
22     printf(" addingup(5)  = %d\n", addingup(5));
23     printf(" addingup(-2) = %d\n\n", addingup(-2));
24
25     /* Reset addingup() */
26     puts(" Reset addingup()");
27     resetValue = addingup(0);
28     addingup(-resetValue);
29     printf(" addingup(0) returns actual value 'static int sum' =%3d\n\n",
30            addingup(0));
31
32     return 0;
33 }
34
35 int f1(void)
36 {
37     int i = 0; /* local variable created on the stack,
38                   every time this function is called */
39     i++;
40     return i;
41 }
42
43 int f2(void)
44 {
45     static int i = 0; /* static local variable created on the heap,
46                          initial value: at first function call */
47     i++;
48     return i;
49 }
50
51 int addingup(int delta)
52 {
53     static int sum = 0;
54     sum += delta;
55     return sum;
56 }
```

The output shows that *f*1() always returns the same result after each invocation. *f*2() and *addingup*() return the updated value of a static local variable after each invocation:

```
f1() called 4 times, returned values:  1  1  1  1
f2() called 4 times, returned values:  1  2  3  4

addingup(0)  = 0
addingup(5)  = 5
addingup(-2) = 3

Reset addingup()
addingup(0) returns actual value 'static int sum' =  0
```

1.9 Exercises

Exercise1

Implement a function for displaying a hex dump of the bytes of a *long* value. Invoke this function several times with different input values in *main*(). Prototype:

```
void hexdumpLong(long data);
```

Exercise2

Implement a function changing the two *double* input values to their average value. Invoke this function several times with different input values in *main*(). Prototype:

```
void average2D(double *pD1, double *pD2);
```

Do not implement the printing of the resulting values in this function because this will considerably limit the re-usability of this function. Print the resulting values in the updated arrays in *main*().

By using an empty body in the definition of the function, the code becomes compilable. The preprocessor #*warning* directive is added to the empty body of the function for showing at compile time that this function still needs to be implemented. This directive is not part of the C standard, but is supported by the *gcc* compiler. Using this directive will limit the portability of the code.

Listing 1.9: average2D/main.c

```
1  #include <stdio.h>
2
3  void average2D(double *pD1, double *pD2);
4
5  int main(void)
6  {
7     double d1 = 2.0;
8     double d2 = 3.0;
9
10    average2D(&d1, &d2);
11    printf(" d1 = %lf   d2 = %lf\n", d1, d2);
12
13    return 0;
14 }
15
16 void average2D(double *pD1, double *pD2)
17 {
18 #warning function average2D() needs to be implemented!
19 }
```

The resulting value for *d*1 should be 2.5 and for *d*2 should also be 2.5.

Exercise3

Implement a function changing the content of a *double* array for all elements to the average value of all the elements in the array. Invoke this function several times with different input values in *main*(). Prototype:

void averageDdata(*double data*[], *int size*);

· Why do we need to pass the size of the array to this function?

Listing 1.10: averageDdata/main.c

```
1  #include <stdio.h>
2
3  #define DATA_SIZE 4
4
5  void averageDdata(double data[], int size);
6
7  int main(void)
8  {
9     double sensorData[DATA_SIZE] = {1.0, 2.0, 3.0, 4.0};
10
11    averageDdata(sensorData, DATA_SIZE);
```

```
12
13     for (int i = 0; i < DATA_SIZE; i++)
14     {
15        printf(" %lf ", sensorData[i]);
16     }
17     puts("");
18
19     return 0;
20  }
21
22  void averageDdata(double data[], int size)
23  {
24  #warning function averageDdata() needs to be implemented!
25  }
```

The resulting values in *sensorData* should be: 2.5, 2.5, 2.5 and 2.5.

Print these resulting values in *main*(), not in the function *averageDdata()* because this will limit the re-usability of this function.

Any code of your own you haven't looked at for six months might as well have been written by someone else.

Eagleson's law

2

Bool type

2.1 Introduction

The boolean type is a fundamental data type in many high-level programming languages such as C, C++, C#, Java and Python.

In ANSI C a boolean value is represented by an integer type. If this value equals 0 this is considered *false*. All non-zero values are considered *true*. The smallest addressable thing in C is a *char*, a single bit is not addressable. The size of a boolean variable can not be smaller than a byte (is a *char*).

The *bool* type (boolean type) is not a native type in ANSI C. C99 and C11 standards have a *bool* type available. A variable of this type can have values *true* and *false*, and all logical operators can be applied in logical expressions.

2.2 Using the bool type to improve code quality

When you use the *bool* type (sized 1 byte) it is clear that the value is *false* or *true*. When you use an *int* for a boolean it is no longer clear in your code that it should be valued 0 or 1. Using the *bool* type will improve the code's quality and the readability.

To avoid the usage of *int* typed variables for implementing a logical value, you need to include *stdbool.h*. The native *bool* type is actually called *_Bool*, a standard library macro defined in *stdbool.h* translates this to *bool*.

The next code example shows the usage of *bool* and two functions returning a *bool* typed value. The input of the function *stringIsAllUppercase* is a *const char str[]*. The keyword *const* is used by the compiler to check if the contents of *str* is not changed in the body of this function. This causes no run-time overhead because this is a compile-time check.

Listing 2.1: bool/main.c

```
1  /* bool type in C language: it is a logical type introduced from C99.
2   * It is the replacement of _Bool type. It takes just 1 byte to store
3   * either true or false. 1 is stored when true is assigned and 0 is
4   * stored when false is assigned.
5   *
6   * A pre-processor statement #include<stdbool.h> is added to the program
7   * to use the bool type.
8   *
9   * %d is the format specifier used to represent a bool in printf()
10  * and scanf().
11  */
12
13 #include <ctype.h>
14 #include <stdbool.h>
15 #include <stdio.h>
16 #include <string.h>
17
18 bool isEven(int x);
19 bool stringIsAllUppercase(const char str[]);
20
21 int main(void)
22 {
23    int a = 2;
24    int b = 10;
25    char text1[10] = "ABCD";
26    char text2[10] = "abCD";
27
28    bool isOK = false;
29    bool isSmallerThan = a < b; /* gives 1 == true */
30
31    printf("  Size of bool is: %lu byte(s)\n\n", sizeof(bool));
32
33    printf("  (a < b) == (%d < %d) == %d\n", a, b, isSmallerThan);
34
35    printf("  isOK = %s\n\n", isOK ? "true" : "false");
36
37    for (int i = 10; i < 16; i++)
38    {
39       if (!isEven(i))
40       {
41          printf("  %d is odd\n", i);
42       }
43    }
44    printf("\n");
45
46    if (stringIsAllUppercase(text1))
47    {
48       printf("  String '%s' contains only uppercase characters\n\n",
```

```
49              text1);
50    }
51    if (!stringIsAllUppercase(text2))
52    {
53       printf("  String '%s' contains not only uppercase characters\n\n",
54              text2);
55    }
56
57    return 0;
58 }
59
60 bool isEven(int x)
61 {
62    return (x % 2 == 0);
63 }
64
65 bool stringIsAllUppercase(const char str[])
66 {
67    bool result = true;
68    size_t size = strlen(str);
69
70    for (size_t i = 0; i < size; i++)
71    {
72       if (!isupper(str[i]))
73       {
74          result = false;
75          /* break will immediately terminate the for-loop */
76          break;
77       }
78    }
79    return result;
80 }
```

Output:

```
Size of bool is: 1 byte(s)

(a < b) == (2 < 10) == 1
isOK = false

11 is odd
13 is odd
15 is odd

String 'ABCD' contains only uppercase characters

String 'abCD' contains not only uppercase characters
```

Two different naming conventions for pointer function parameters:

bool stringIsAllUpperCase(const char str[])

*bool stringIsAllUpperCase(const char *pStr)*

If you know that a function pointer parameter is pointing to an array, you could prefer the first one.

2.3 Exercises

Exercise1

Implement a function returning a *bool* typed result for comparing the content of two same sized *int* arrays: *arr*1 and *arr*2. Use a for-loop and a break statement (see code example *bool stringIsAllUpperCase*(*const char str*[])).

Test this function in *main*(). Prototype:

bool compareIntArrays(*const int arr*1[], *const int arr*2[], *int size*);

Example of calling this function and printing the boolean value as *true* or *false* text:

```
1 #define DATA_SIZE 4
2
3 int data1[DATA_SIZE] = {1, 2, 3, 4};
4 int data2[DATA_SIZE] = {1, 2, 3, 4};
5
6 bool arraysAreEqual = compareIntArrays(data1, data2, DATA_SIZE);
7
8 printf("Arrays are equal = %s\n", arraysAreEqual ? "true" : "false");
```

Controlling complexity is the essence of computer programming.

Brian Kernigan

3

Command line parameters

3.1 Introduction

A program can be started up by the graphical user interface of an operating system, a text based console or an IDE. A text-based console will show you a prompt for entering a command. "A command line is the space to the right of the command prompt on an all-text display mode on a computer monitor in which a user enters commands and data. It provides a means of communication between a user and a computer that is based solely on textual input and output. An all-text display mode also referred to as a command line interface (CLI), can be provided by both a console and a terminal window [7]."

It is possible to pass a number of strings from the command line to C programmes when they are started by some command. These strings separated by spaces or tabs are called command line arguments.

3.2 Command line argument handling

You can not longer use *int main*(*void*) if you want to use command line arguments. The command line arguments are handled by the function parameters *argc* (argument count) and *argv* (argument vector) of *main*():

```
int main(int argc, char *argv[])
```

Or:

```
int main(int argc, char **argv)
```

There is no fundamental difference between *char *argv[]* and *char **argv*.

argv is an array (vector) holding *char* pointers (*char**) to every command line argument (strings) stored in memory:

- *argv* pointer to an array with string pointers (*char**), pointer to a pointer (*char***)
- *argv*[0] pointer (*char**) to the name of the full path of the executable
- *argv*[*i*] pointer (*char**) to *i*-th command line argument, *i* <= *argc*
- *argv*[*argc*] pointer (*char**) has the value NULL
- *argv*[1][2] character (*char*), third character in *argv*[1]

You should always check in the code the number of required command line arguments before you start using them. If the number of arguments is incorrect you should show a usage message on *stderr* (standard error stream, unbuffered output) providing information about the required arguments (syntax description).

If the number of expected arguments is correct you should also code a further check of every command line argument. For example, if an argument represents a valid integer value in the correct range, represents a valid file name with a correct extension or represents a valid MAC-address. Robust programming is necessary to avoid *garbage in, is garbage out* (nonsense input will produce nonsense output).

Because the command line parameters are strings (array of characters), you need to use functions for converting them into integer or double typed values. These functions are available in the standard C library, see String conversions.

3.3 Code examples using argc and argv

Next code example shows how to print all entered command line parameter values in a for-loop. The for-loop uses *argc* to check the boundaries of *argv*. If the number of arguments is not correct (*argc* $<$ 2) the program will stop by calling *exit*().

Note that *exit*() takes an integer argument, 0 usually means your program has completed successfully, and non-zero values are used as error codes. Two standard error codes are available: *EXIT_SUCCESS* and *EXIT_FAILURE* as defined in *stdlib.h*. Any other error code values must be defined by the developer. These error codes could be mentioned in the usage message or the documentation of the program.

Listing 3.1: echo-argcargv-forloop/main.c

```
1  /* Print all command line items one by one to screen (stdout).
2   * Example with 3 additional commandline parameters:
3   *
4   *    >./echo-argcargv-forloop file.txt 3 "input data for file"
5   *
6   * In QtCreator input commandline parameters:
7   *
8   *    Projects Build&Run, Run, Arguments
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14
15 int main(int argc, char *argv[])
16 {
17    /* Always check the number of required arguments */
18    if (argc < 2)
19    {
20       /* Error messages to stderr */
21       fprintf(stderr,
22               "\tERROR %s: \n"
23               "\tUSAGE: expected 1 or more commandline parameters\n\n",
24               argv[0]);
25       exit(EXIT_FAILURE);
26    }
27
28    /* Print all command line arguments */
29    for (int i = 1; i < argc; i++)
30    {
31       printf("- argv[%d]: %s\n", i, argv[i]);
32    }
33    puts("\n");
34
35    /* Print the third character of all command line arguments if available
36     */
37    for (int i = 1; i < argc; i++)
38    {
39       /* Check for availability third character */
40       if (strlen(argv[i]) > 2)
41       {
42          printf("- argv[%d][2]: %c\n", i, argv[i][2]);
43       }
44    }
45    puts("\n");
46
47    return 0;
48 }
```

If no command line arguments are given *argc* == 1, because the name of the program qualifies the first argument, the output will be an usage error message:

```
USAGE: expected 1 or more commandline parameters
```

The next command line example contains a quoted argument. A quoted argument is passed as one argument:

```
file.txt 3 "input data for file"
```

This command line has four arguments (*argc* == 4). *argv*[0] is not shown:

```
- argv[1]: file.txt
- argv[2]: 3
- argv[3]: input data for file

- argv[1][2]: l
- argv[3][2]: p
```

Next code example shows the way of printing all entered command line parameter values in a while-loop. The while-loop tests for the NULL value of *argv*[*i*]. If the number of arguments is incorrect (*argc* < 2) the program will stop by calling *exit*().

Listing 3.2: echo-argcargv-whileloop/main.c

```
1  /* Print all command line items one by one to screen (stdout).
2   * Example with 3 additional commandline parameters:
3   *
4   *    >./echo-argcargv-whileloop file.txt 123 6*2-100
5   *
6   * In QtCreator input commandline parameters:
7   *
8   *    Projects Build&Run, Run, Arguments
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int main(int argc, char *argv[])
15 {
16    int i = 0;
17
18    /* Always check the number of required arguments */
19    if (argc < 2)
20    {
```

```
21        /* Error messages to stderr */
22        fprintf(stderr,
23                "\tERROR %s: \n"
24                "\tUSAGE: expected 1 or more commandline parameters\n\n",
25                argv[0]);
26        exit(EXIT_FAILURE);
27     }
28
29     /* Print all command line arguments */
30     i = 1;
31     while (argv[i] != NULL)
32     {
33        printf("- argv[%d]: %s\n", i, argv[i]);
34        i++;
35     }
36     printf("- argv[%d]: %s\n", i, argv[i]);
37     puts("\n");
38
39     return 0;
40  }
```

The next command line arguments are given:

```
file.txt 123 6*2-100
```

Output:

```
- argv[1]: file.txt
- argv[2]: 123
- argv[3]: 6*2-100
- argv[4]: (null)
```

3.4 String conversions

There are several standard string conversion functions available in C. A string contains ASCII characters and is terminated by a null character. We need to include *stdlib.h*. Two examples will show the prototypes of C standard library functions for conversion of a string to an *int* and a *double* value.

- 1. Conversion from an ASCII string to an *int* value (prototype):

```
int atoi(const char *str);
```

If the string contains a decimal place, the number will be truncated. If no valid conversion could be performed, it returns 0.

- 2. Conversion from an ASCII string to a *double* value (prototype):

```
double atof(const char *str);
```

If the string contains a decimal place, the number will not be truncated. If no valid conversion could be performed, it returns 0.0.

The next example shows these two conversion functions using different input strings.

Listing 3.3: atoiatof/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6     printf(
7        "-- int atoi(const char *str);\n"
8        "-- Printing integers:\n");
9
10    printf("   123    => %8d \n", atoi("123"));
11    printf("   123.4  => %8d \n", atoi("123.4"));
12    printf("   123abc => %8d \n", atoi("123abc"));
13    printf("   abc123 => %8d \n", atoi("abc123"));
14    puts("");
15
16    printf(
17       "-- double atof(const char *str);\n"
18       "-- Printing doubles:\n");
19
20    printf("   123.012    => %11lf \n", atof("123.012"));
21    printf("   123.012abc => %11lf \n", atof("123.012abc"));
22    printf("   123        => %11lf \n", atof("123"));
23    printf("   abc123     => %11lf \n", atof("abc123"));
24    puts("");
25
26    return 0;
27 }
```

The output shows formatted *int* values:

```
 -- int atoi(const char *str);
 -- Printing integers:
    123    =>      123
    123.4  =>      123
    123abc =>      123
    abc123 =>        0
```

And the output shows formatted *double* values:

```
-- double atof(const char *str);
-- Printing doubles:
   123.012    =>  123.012000
   123.012abc =>  123.012000
   123        =>  123.000000
   abc123     =>    0.000000
```

3.5 Exercises

Exercise1

Write a program that adds up all command line arguments representing a valid integer value without a preceding + or − sign.

Use *atoi*() to convert from an ASCII string to an integer value. If no arguments are given an appropriate error message needs to be given. Use the *stderr* stream for this error message.

Use the next function for checking if a string represents a valid positive integer value without a preceding + of − sign. Implementation:

```
1  bool isInt(const char str[])
2  {
3     int i = 0;
4     bool isInteger = true;
5
6     while (str[i] != '\0')
7     {
8        if (!isdigit(str[i]))
9        {
10          isInteger = false;
11          break;
12       }
```

```
13        i++;
14    }
15    return isInteger;
16 }
```

This function uses *isdigit*() for testing every character in the string *str* in a loop until a character is not a digit or the end of the string is reached (test for the null character). If this condition occurs the loop is terminated by a *break statement*.

Use *isInt*() in *main*() for checking in a loop if all command line arguments (*argv*[*i*]) are valid positive integers. This style of programming should prevent abnormal behaviour or crashing of the program if an input is not valid. If one of the arguments is not an integer, an appropriate usage message needs to be given before terminating the program with *exit*(). Use the *stderr* stream for this message.

Why shouldn't you implement the printing of result messages in the *isInt*() function?

Exercise2

Update the function *isInt*() to check if a string represents a valid integer value possibly preceded by only one + or − sign. Some examples:

```
Valid integer strings, isInt() returns true:

   "0"
   "123"
   "-123"
   "+123"
```

```
Invalid integer strings, isInt() returns false:

  isInt("a123")
  isInt("123a")
  isInt("--123")
  isInt("+-123")
  isInt("-12+3")
  isInt("123+")
  isInt("1-23")
  isInt("+++123")
  isInt("+")
  isInt("-")
  isInt("")
```

Use the showed valid and invalid integer string value examples for testing the updated function *isInt*() in *main*().

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

Rick Osborne

4

Complex numbers

4.1 Introduction

A complex number is a number that can be expressed in the form $a + bi$, where a and b are real numbers and i is the imaginary unit which satisfies the equation $i^2 = -1$. In this expression, a is the real part and b is the imaginary part of the complex number. Complex numbers play an important role in a variety of engineering fields.

The *complex* type is a fundamental data type in many high-level programming languages. C99 and C11 standards have a complex type available. For using this type you must include *complex.h*.

A complex variable is declared by adding *complex* to the data type name separated by a space.

- *float complex cf;*
- *double complex cd*;
- *long double complex cld*;
- *int complex ci*;

The standard arithmetic operators +, −, ∗ and / can be used for complex numbers in expressions. The macro constant *I* is the imaginary unit constant *i*. There is no format specifier for complex (structured) data types available that can be used in the *printf*() format strings.

4.2 Built-in functions for complex number arithmetic

You must be careful when using the mathematical functions defined in *math.h* as these do not conform with complex algebra. Include *complex.h*.

Some built-in functions:

- *creal*() to obtain the real part of a complex number.
- *cimag*() to obtain the imaginary part of a complex number.
- *cabs*() to calculate the absolute value of a complex number.
- *cpow*() to calculate the complex power of a *double complex* type.
- *csqrt*() to calculate the complex square root of a *double complex* type.

Once you have declared some complex variables and assigned values to them, you can do the standard mathematical operations as demonstrated in the next code example. Only *double complex* types are used.

In the format string for printing complex numbers you use a + sign after the % for always printing the leading + sign.

In the text output of the next code example (lines 53-56) the caret ^ symbol is used to denote exponentiation.

```
    1+5i^3      equals      1 + 5 * i * i * i
```

Listing 4.1: complex/main.c

```
1  /* Complex numbers: added in C99 and C11 */
2
3  #include <complex.h>
4  #include <stdio.h>
5
6  int main(void)
7  {
8     double complex z1 = 1.0 + 3.0 * I;
9     double complex z2 = 1.0 - 4.0 * I;
10    double complex z3 = 0.0;
11    double complex z4 = 0.0;
12    double complex z5 = 0.0;
13
14    printf("Working with complex numbers\n\n");
15
16    printf(
```

```
17        "Starting values:\n"
18        "z1               = %6.2lf%+6.2lfi\n"
19        "z2               = %6.2lf%+6.2lfi\n",
20        creal(z1), cimag(z1), creal(z2), cimag(z2));
21
22     double complex sum = z1 + z2;
23     printf("z1 + z2          = %6.2f%+6.2fi\n", creal(sum), cimag(sum));
24
25     double complex difference = z1 - z2;
26     printf("z1 - z2          = %6.2f%+6.2fi\n", creal(difference),
27            cimag(difference));
28
29     double complex product = z1 * z2;
30     printf("z1 x z2          = %6.2f%+6.2fi\n", creal(product),
31            cimag(product));
32
33     double complex quotient = z1 / z2;
34     printf("z1 / z2          = %6.2f%+6.2fi\n", creal(quotient),
35            cimag(quotient));
36
37     double complex conjugate = conj(z1);
38     printf("conjugate of z1  = %6.2f%+6.2fi\n\n", creal(conjugate),
39            cimag(conjugate));
40
41     z1 = I * I;
42     printf("z1 = I * I       = %6.2f%+6.2fi\n", creal(z1), cimag(z1));
43
44     z1 *= I;
45     printf("z1 = I * I * I   = %6.2f%+6.2fi\n", creal(z1), cimag(z1));
46
47     /* Multiplying conjugates */
48     z4 = 1 + 2 * I;
49     z5 = 1 - 2 * I;
50     printf("z4 * z5          = %6.2f%+6.2fi\n\n", creal(z4 * z5),
51            cimag(z4 * z5));
52
53     /* ^ raise to the power
54      * Calculate 1 + 5i^3 = ? */
55     z1 = 1 + 5 * cpow(I, 3);
56     printf("1 + 5i^3         = %6.2f%+6.2fi\n\n", creal(z1), cimag(z1));
57
58     /* Calculate (2 + i) - (2i - 3) = ? */
59     z1 = 2 + I;
60     z2 = 2 * I - 3;
61     z3 = z1 - z2;
62     printf(
63        "(2 + i)-(2i - 3) = %6.2f%+6.2fi  \n"
64        "                 = in polar form r = %.2lf theta = %.2lf\n\n",
65        creal(z3), cimag(z3), cabs(z3), carg(z3));
66
67     return 0;
68  }
```

The output of several expressions containing complex variables and values:

```
Working with complex numbers

Starting values:
z1               =   1.00 +3.00i
z2               =   1.00 -4.00i
z1 + z2          =   2.00 -1.00i
z1 - z2          =   0.00 +7.00i
z1 x z2          =  13.00 -1.00i
z1 / z2          =  -0.65 +0.41i
conjugate of z1  =   1.00 -3.00i

z1 = I * I       =  -1.00 +0.00i
z1 = I * I * I   =  -0.00 -1.00i
z4 * z5          =   5.00 +0.00i

1 + 5i^3         =   1.00 -5.00i

(2 + i)-(2i - 3) =   5.00 -1.00i
                 = in polar form r = 5.10 theta = -0.20
```

Additional functions for processing complex numbers are available in *tgmath.h*.

4.3 Exercises

Exercise1

Calculate and print the complex result of the following mathematical formulas:

- $result1 = (-3 - i)(5 - 2i)$
- $result2 = \dfrac{1 - i}{i + 3}$

The function of good software is to make the complex appear to be simple.

Grady Booch

5

Void pointers and callback functions

5.1 Introduction

In general, pointers are typed. For some important reasons you sometimes must consider the usage of untyped pointers because if you want for example to implement functions for processing all kind of input data these untyped pointers are necessary. Otherwise you would have to copy-paste a lot of code and only change the pointer types to the types you want to use.

5.2 Generic pointers or void pointers

Void pointers are *generic pointers* : they can point to any data type. Void pointers do not interpret the data they are pointing to. If you want to dereference a void pointer you must always apply a type cast to the specific pointer type. A programmer is responsible for implementing the correct type cast because the compiler can not check this. Pointer arithmetic is not possible for void pointers.

Listing 5.1: voidPointers/main.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
```

```
5     void *pGeneric = NULL;
6
7     int iVar = 1;
8     pGeneric = &iVar;
9     printf(" pGeneric = %p points to %d\n", pGeneric, *(int *)pGeneric);
10
11    double dVar = 2.2;
12    pGeneric = &dVar;
13    printf(" pGeneric = %p points to %lf\n", pGeneric, *(double *)pGeneric);
14
15    char cVar = 'c';
16    pGeneric = &cVar;
17    printf(" pGeneric = %p points to %c\n", pGeneric, *(char *)pGeneric);
18
19    return 0;
20 }
```

5.3 Function pointers and callback functions

The name of a function is a pointer to this function. We are able to use all kinds of pointers as an input parameter for functions. This means that functions are also able to use a function pointer in the parameter list. This function pointer enables a function for calling an external function by this pointer. This external function is called a *callback function* . Callback functions are not only available in C but also in many other programming languages.

Function pointers are typed pointers. A function pointer may be typed like:

$$double(*)(int, int)$$

in case where we point to a function that takes two *int* typed values and returns an *double* typed value. But, when declaring a function pointer, its identifier (pointer name) is not placed after the type, like:

```
double (*)(int, int) fpointer;        /* not correct!! */
```

Instead, you must write the function pointer identifier *fpointer* in the middle:

```
double (*fpointer)(int, int);
```

You do not have to dereference function pointers. Dereferencing a function pointer just evaluates back to the function pointer. As far as the compiler is concerned *fpointer* and *(*fpointer)* are identical. It is possible to call a function by its function pointer.

Listing 5.2: function-pointers/main.c

```
1  #include <stdio.h>
2
3  int add(int a, int b);
4  int multiply(int a, int b);
5
6  int main(void)
7  {
8     int result = 0;
9     /* function pointer variable declaration */
10    int (*fpointer)(int, int) = add;
11
12    printf("Address 'add()'      = %p\nValue   'fpointer'   = %p\n", add,
13           fpointer);
14
15    /* function call by function pointer */
16    result = fpointer(2, 5);
17    printf("result = fpointer(2, 5);  result = %d\n\n", result);
18
19    /* update function pointer variable */
20    fpointer = multiply;
21    printf("Address 'multiply()' = %p\nValue   'fpointer'   = %p\n",
22           multiply, fpointer);
23
24    /* function call by function pointer */
25    result = fpointer(2, 5);
26    printf("result = fpointer(2, 5);  result = %d\n", result);
27
28    return 0;
29 }
30
31 int add(int a, int b)
32 {
33    return a + b;
34 }
35
36 int multiply(int a, int b)
37 {
38    return a * b;
39 }
```

Output:

```
Address 'add()'      = 0x561be8abc6e1
Value   'fpointer'   = 0x561be8abc6e1
result = fpointer(2, 5);  result = 7

Address 'multiply()' = 0x561be8abc6f5
Value   'fpointer'   = 0x561be8abc6f5
result = fpointer(2, 5);  result = 10
```

The next code example shows that you can register a function that will be called if a program terminates either via *exit*() or via a return from the program's *main*(). Functions so registered are called in the reverse order of their registration. Prototype:

```
int atexit(void (*function)(void));
```

For using *atexit*() you must include *stdlib.h*.

Listing 5.3: atexit/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void shutdown(void);
5  int calculate(int x);
6
7  int main(void)
8  {
9     printf("Function set to be executed on exit (if program terminates)\n");
10    atexit(shutdown);
11
12    for (int input = -5; input < 20; input++)
13    {
14       printf("-- Calculating %d\n", calculate(input));
15    }
16
17    return 0;
18 }
19
20 void shutdown(void)
21 {
22    fprintf(stderr, "Shutting down\n");
23 }
24
25 int calculate(int x)
26 {
27    if (x < -10 || x > 10)
28    {
29       fprintf(stderr, "ERROR calculate(): '%d' is out of range\n", x);
30       exit(EXIT_FAILURE);
31    }
```

```
32     return x * x * x * x;
33 }
```

The output shows the invocation of the *exit*() callback function before shutting down:

```
Function set to be executed on exit (if program terminates)
-- Calculating 625
-- Calculating 256
-- Calculating 81
-- Calculating 16
-- Calculating 1
-- Calculating 0
-- Calculating 1
-- Calculating 16
-- Calculating 81
-- Calculating 256
-- Calculating 625
-- Calculating 1296
-- Calculating 2401
-- Calculating 4096
-- Calculating 6561
-- Calculating 10000
ERROR calculate(): '11' is out of range
Shutting down
```

The next code example shows two functions each using two callback functions and the use of *typedef* for giving a callback function type a new name.

Listing 5.4: callback/main.c

```
1  /* Function pointers and callback functions */
2
3  #include <math.h>
4  #include <stdio.h>
5
6  typedef double (*callbackMath_t)(double);
7
8  int sqr(int x);
9  int redouble(int x);
10 int halve(int x);
11
12 int compose(int x, int (*pf1)(int), int (*pf2)(int));
13 double composeMath(double x, callbackMath_t f1, callbackMath_t f2);
14
15 int main(void)
16 {
```

```
17     printf("sqr(sqr(1)) = %d\n", compose(1, sqr, sqr));
18     printf("sqr(redouble(1)) = %d\n", compose(1, redouble, sqr));
19     printf("halve(redouble(2) = %d\n", compose(2, redouble, halve));
20     printf("sin(sqrt(1.0)) = %lf\n", composeMath(1.0, sqrt, sin));
21
22     return 0;
23 }
24
25 int sqr(int x)
26 {
27     return x * x;
28 }
29
30 int redouble(int x)
31 {
32     return x + x;
33 }
34
35 int halve(int x)
36 {
37     return x / 2;
38 }
39
40 int compose(int x, int (*pf1)(int), int (*pf2)(int))
41 {
42     return pf2(pf1(x));
43 }
44
45 double composeMath(double x, callbackMath_t f1, callbackMath_t f2)
46 {
47     return f2(f1(x));
48 }
```

Output:

```
sqr(sqr(1)) = 1
sqr(redouble(1)) = 4
halve(redouble(2) = 2
sin(sqrt(1.0)) = 0.841471
```

5.4 Sorting data

Sorting means arranging data in an ordered sequence. The function *qsort*() (quick sort, is a sorting strategy) can sort an array of any kind of data. For sorting of data, you need to be able to compare the data elements in the array. We need a *generic function* as a callback function for *qsort*() to do the necessary comparison. This generic

callback function uses two *void* pointers to point to two different data elements in the array.

This function must be implemented by the developer of the code. Inside this function, the correct type casts for these pointers must be applied for the comparison. This function must return −1 or a value < 0 if the first element is ”less than” the second, 1 or a value > 0 if the first element is ”greater than” the second and 0 if the two elements are equal. The array contents will be sorted in ascending order. Ascending order is often the standard order for sorting because the standard order of numbers is ascending (1, 2, 3, 4, ...).

Prototype *qsort*() function:

*void qsort(void * base, size_t num, size_t size,*
*int (*compar)(const void*, const void*));*

- *base* is the generic pointer to the first element of the array to be sorted.
- *num* is the number of array elements.
- *size* the size in bytes of each element in the array.
- *compar* is the function pointer to a necessary callback function for comparing two data elements.

The next code example shows the usage of *qsort*() and an appropriate callback function for sorting the contents of an array containing integers (simple data type).

Listing 5.5: qsort-ints/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE 5
5
6  int compareInts(const void *pInt1, const void *pInt2);
7
8  int main(void)
9  {
10    int i = 0;
11    int data[SIZE] = {-1, 6, 5, -2, -4};
12
13    puts("Initial contents 'data' array:");
14    for (i = 0; i < SIZE; i++)
15    {
16       printf("%d ", data[i]);
17    }
18    puts("\n");
19
```

```
20     puts("Sorted data[1] ... data[3]:");
21     qsort(data + 1, 3, sizeof(int), compareInts);
22     for (i = 0; i < SIZE; i++)
23     {
24        printf("%d ", data[i]);
25     }
26     puts("\n");
27
28     puts("Full array 'data' sorted:");
29     qsort(data, SIZE, sizeof(int), compareInts);
30     for (i = 0; i < SIZE; i++)
31     {
32        printf("%d ", data[i]);
33     }
34     puts("");
35
36     return 0;
37  }
38
39  int compareInts(const void *pInt1, const void *pInt2)
40  {
41     int i1 = *(const int *)pInt1;
42     int i2 = *(const int *)pInt2;
43
44     if (i1 < i2)
45     {
46        return -1;
47     }
48     if (i1 > i2)
49     {
50        return 1;
51     }
52     return 0;
53  }
```

Output after two sorting steps of the array *data*:

```
Initial contents 'data' array:
-1 6 5 -2 -4

Sorted data[1] ... data[3]:
-1 -2 5 6 -4

Full array 'data' sorted:
-4 -2 -1 5 6
```

The next code example shows the usage of *qsort*() and an appropriate callback function for sorting the contents of an array containing structs (complex structured data type).

Listing 5.6: qsort/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE 5
5
6  typedef struct
7  {
8     int iData;
9     double dData;
10 } data_t;
11
12 /* comparison functions */
13 int compareInts(int i1, int i2);
14 int compareDoubles(double d1, double d2);
15 int compareData(const void *pD1, const void *pD2);
16
17 int main(void)
18 {
19    data_t data[SIZE] = {
20      {2, 10.1},
21      {2, 30.2},
22      {2, 20.3},
23      {-2, 100.20},
24      {-2, 100.10}
25    };
26
27    printf("---- Not sorted:\n");
28    for (int i = 0; i < SIZE; i++)
29    {
30       printf("data[%d] = {%2d, %7.3lf}\n", i, data[i].iData, data[i].dData);
31    }
32    printf(" \n");
33
34    /* Sorting algorithm: quick sort */
35    /* compareData is a function pointer to a callback function */
36    qsort(data, SIZE, sizeof(data_t), compareData);
37
38    printf("-------- Sorted:\n");
39    for (int i = 0; i < SIZE; i++)
40    {
41       printf("data[%d] = {%2d, %7.3lf}\n", i, data[i].iData, data[i].dData);
42    }
43    printf("\n");
44
45    return 0;
46 }
47
48 int compareInts(int i1, int i2)
49 {
50    if (i1 < i2) return -1;
```

```
51    if (i1 > i2) return 1;
52    return 0;
53 }
54
55 int compareDoubles(double d1, double d2)
56 {
57    if (d1 < d2) return -1;
58    if (d1 > d2) return 1;
59    return 0;
60 }
61
62 /* callback function for qsort */
63 int compareData(const void *pD1, const void *pD2)
64 {
65    const data_t *pData1 = (const data_t*)pD1;
66    const data_t *pData2 = (const data_t*)pD2;
67
68    if ((pData1->iData) == (pData2->iData))
69    {
70       return compareDoubles(pData1->dData, pData2->dData);
71    }
72    return compareInts(pData1->iData, pData2->iData);
73 }
```

Output not sorted (initial values) and sorted:

```
---- Not sorted:
data[0] = { 2,  10.100}
data[1] = { 2,  30.200}
data[2] = { 2,  20.300}
data[3] = {-2, 100.200}
data[4] = {-2, 100.100}

-------- Sorted:
data[0] = {-2, 100.100}
data[1] = {-2, 100.200}
data[2] = { 2,  10.100}
data[3] = { 2,  20.300}
data[4] = { 2,  30.200}
```

5.5 Exercises

Exercise1

Write a program for sorting an array of zip codes (postal codes) using *qsort*(). A zip code must be implemented in a struct. Use a *typedef* for the type name of this struct.

Example Dutch postal code:

```
1 /* Dutch postal code, example 1200AB */
2
3 typedef struct
4 {
5    int number;
6    char twoChars[3];  /* string, needs additional '\0' */
7 } zipcode_t;
```

Implement an appropriate necessary compare function for two of these structs containing zip codes. Prototype:

```
int compareZipCodes(const void * pZC1, const void * pZC2);
```

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

6 Singly linked lists

6.1 Introduction

Organising data for processing is an essential step in the development of computer programs. Arrays and linked lists are important standard *data structures*. A data structure organises data and provides functions for accessing and manipulating the contained data.

An Singly Linked List (SLL) is a linear collection of sequential data elements, called nodes. Each node must be implemented in a C struct containing the data and a pointer to the next node. An SLL is a data structure holding a group of nodes which together represent a sequence. An SLL can only be traversed by a pointer in the forward direction.

An SLL can be used for implementing a *stack* (LIFO) and a *queue* (FIFO). The SLL implementation must use dynamic memory allocation because the code must be able to add and remove nodes at runtime. Dynamically allocated memory comes from a pool of memory known as the *heap* (free memory). In C you use the function *malloc*() for allocating memory and *free()* for releasing the allocated memory. The latter function only needs the pointer you got from *malloc*(), the size of allocated memory is cached by the internal system memory management functions.

Prototypes:

```
void * malloc(size_t size);
void free(void *ptr);
```

The function *free()* is a generic function because it is uses an untyped pointer.

Some important coding guidelines for avoiding undefined behaviour and memory leakage are:

- Always check if allocation of memory by *malloc*() has succeeded. The returned pointer value should not equal *NULL*. But, if you are using the (Embedded) Linux operating system this is not guaranteed. Solving this problem is outside the scope of this book how to manage this problem.
- Do not free the same block of dynamic allocated memory more than once.
- Do not call *free()* on a pointer not returned by *malloc*().
- Calling *free(NULL)* is allowed because it causes no undefined behaviour. When allocated memory space is freed, the pointer used should be assigned the value NULL directly after calling the function *free()*.
- Do not allocate memory in a function using a local pointer without freeing the allocated memory in the same function. Because a local pointer value is not available outside the function for freeing the memory.

6.2 SLL implementation

The pointer *pHead* points to the first node in the sequence of nodes. This pointer owns the SLL and its value should only be altered by the implemented and tested SLL API functions. If ignored, it can result in memory leakage or a crashing program. The last node should always contain a NULL pointer for indicating the end of the SLL.

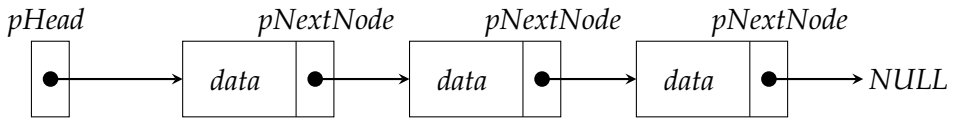
If the SLL is empty *pHead* equals NULL.

A node is defined in *sll.h*:

Listing 6.1: singlyLinkedList/app/sll.h

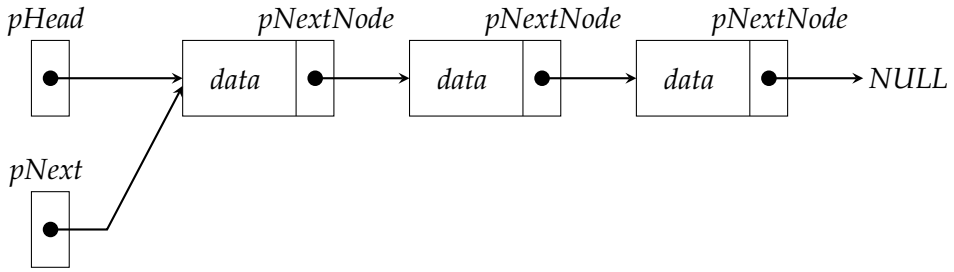
```
1 typedef struct node {
2    int data;
3    struct node *pNextNode;
4 } node_t;
```

The next diagram shows an SLL with 3 nodes.



The next diagrams are the starting point for the implementation of the function *showSLL*() showing the value of a new necessary local pointer *pNext* for traversing the SLL in this function. We must avoid corrupting *pHead*.

The first step is to initialise *pNext* to *pHead*:

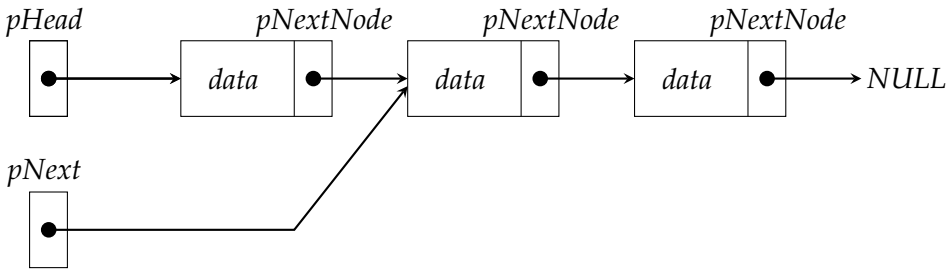


Listing 6.2: singlyLinkedList/app/sll.c

```
1  void showSLL(const node_t *pHead)
2  {
3     const node_t *pNext = pHead;
4
5     if (pHead == NULL)
6     {
7        printf("SLL is empty\n");
8     }
```

- line 3 creates and initialises the local pointer variable *pNext*
- line 5 checks if the SLL is empty

You are going to traverse the SLL until you reach the end of the SLL:



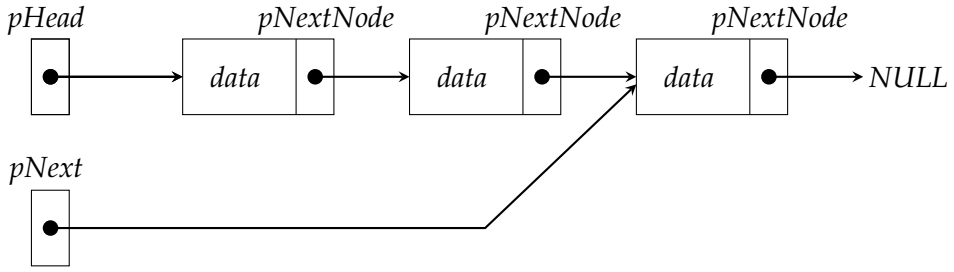
pNext = pNext− > pNextNode;

Listing 6.3: singlyLinkedList/app/sll.c

```
1  void showSLL(const node_t *pHead)
2  {
3     const node_t *pNext = pHead;
4
5     if (pHead == NULL)
6     {
7        printf("SLL is empty\n");
8     }
9     else
10    {
11       while (pNext != NULL)
12       {
13          printf("Node %p:  Data = %d  pNext = %p\n", pNext, pNext->data,
14                 pNext->pNextNode);
15          pNext = pNext->pNextNode;
16       }
17    }
18 }
```

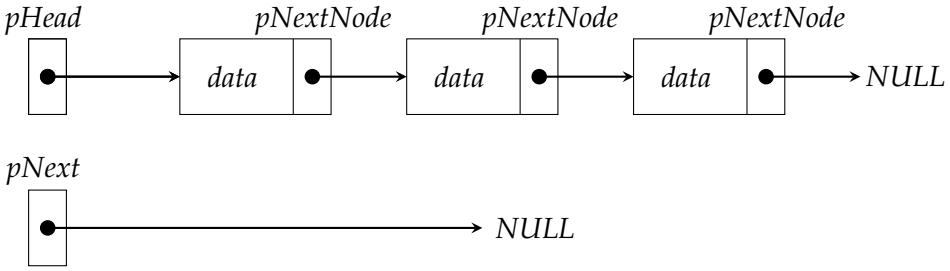
- line 11..16 *pNext* traverses the SLL in a while-loop
- line 15 updates *pNext = pNext-> pNextNode;*

Next step in traversing the SLL:



$pNext = pNext- > pNextNode$

Last step in traversing the SLL (detected end of SLL):



$pNext = pNext- > pNextNode$
$pNext == NULL$ while loop will be ended

The dynamic allocation and deallocation of nodes should be totally hidden by the the SLL API functions (abstraction), so the developer does not need to bother about managing the allocation and deallocation of nodes in *main*(). The data in the SLL is only accessed by its API. The API is behaviour oriented and thus not tell you how the data is organised in memory: *abstract data type* (ADT).

The next code example is split into three separate files (modular programming). This enables us to reuse the SLL module by including the SLL header file *sll.h* in other programs or other modules and by linking the related object file *sll.o*.

Only three SLL API functions are implemented. The next examples are candidate for designing and implementing a function for:

- determining the size of an SLL (number of nodes)
- sorting an SLL
- swapping two SLL's

- concatenating two SLL's
- removing the first node in an SLL
- removing the last node in an SLL

Coding without drawing the related diagrams is very prone to error.

The next code example shows the implementation of four SLL API functions.

Listing 6.4: singlyLinkedList/app/sll.c

```
1  #include "sll.h"
2
3  #include <stdio.h>
4
5  size_t sizeSLL(const node_t *pHead)
6  {
7     const node_t *pNext = pHead;
8     size_t size = 0;
9
10    if (pHead != NULL)
11    {
12       /* Needs to be implemented */
13    }
14    return size;
15 }
16
17 void addSLL(node_t **ppHead, int data)
18 {
19    node_t *pNext = *ppHead;
20    node_t *pNew = (node_t *)malloc(sizeof(node_t));
21
22    /* Check if allocation has succeeded */
23    if (pNew != NULL)
24    {
25       pNew->data = data;
26       pNew->pNextNode = NULL;
27    }
28    if (pNext != NULL)
29    {
30       /* Traverse through every subsequent node in the SLL */
31       while (pNext->pNextNode != NULL)
32       {
33          pNext = pNext->pNextNode;
34       }
35       pNext->pNextNode = pNew;
36    }
37    else
38    {
39       *ppHead = pNew;
40    }
41 }
42
```

```
43 void clearSLL(node_t **ppHead)
44 {
45    node_t *pToBeRemoved = *ppHead;
46    node_t *pNext = NULL;
47
48    while (pToBeRemoved != NULL)
49    {
50       pNext = pToBeRemoved->pNextNode;
51       free(pToBeRemoved);
52       pToBeRemoved = pNext;
53    }
54    *ppHead = NULL;
55 }
56
57 void showSLL(const node_t *pHead)
58 {
59    const node_t *pNext = pHead;
60
61    if (pHead == NULL)
62    {
63       printf("SLL is empty\n");
64    }
65    else
66    {
67       while (pNext != NULL)
68       {
69          printf("Node %p:  Data = %d  pNext = %p\n", pNext, pNext->data,
70                 pNext->pNextNode);
71          pNext = pNext->pNextNode;
72       }
73    }
74 }
```

The next code example shows the usage of the SLL module. The API function *showSLL()* shows important data structure implementation details of the SLL (this violates the ADT approach, but is now used to inform you about the data organisation in memory and the use of pointers). This function should only show the data in the SLL.

Listing 6.5: singlyLinkedList/app/main.c

```
1  /* Singly Linked List: SLL */
2
3  #include "sll.h"
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main(void)
9  {
10    node_t *pHead = NULL; /* Create an empty SLL,
11                             pHead is the 'owner' of the SLL */
12
```

```
13     showSLL(pHead);
14     printf("Size of the SLL = %lu\n", sizeSLL(pHead));
15
16     printf("\nAdd new data to the SLL 0x%p:\n", pHead);
17     addSLL(&pHead, 10);
18     showSLL(pHead);
19     printf("Size of the SLL = %lu\n\n", sizeSLL(pHead));
20
21     printf("Add new data to the SLL 0x%p:\n", pHead);
22     addSLL(&pHead, 20);
23     showSLL(pHead);
24     printf("Size of the SLL = %lu\n\n", sizeSLL(pHead));
25
26     printf("Clear the SLL 0x%p:\n", pHead);
27     clearSLL(&pHead);
28     showSLL(pHead);
29     printf("Size of the SLL = %lu\n\n", sizeSLL(pHead));
30
31     return 0;
32 }
```

Output:

```
SLL is empty
Size of the SLL = 0

Add new data to the SLL 0x(nil):
Node 0x560b9e045270:  Data = 10  pNext = (nil)
Size of the SLL = 0

Add new data to the SLL 0x0x560b9e045270:
Node 0x560b9e045270:  Data = 10  pNext = 0x560b9e045290
Node 0x560b9e045290:  Data = 20  pNext = (nil)
Size of the SLL = 0

Clear the SLL 0x0x560b9e045270:
SLL is empty
Size of the SLL = 0
```

6.3 Tools for finding memory leaks and core dumps

Memory leaks are among the most difficult bugs to detect. *Valgrind* [19] is a memory debugging tool for Linux. It allows you to run a program in *Valgrind*'s own environment that monitors memory usage. It detects the difference between the size of memory allocations and deallocations.

In Linux a core dump is a file containing a memory snapshot of the crashed application (process). You can use the debugger tool *gdb* for showing the contents of the core dump file. The code should be compiled with the debugging option *-g*.

Set the core file limit:

```
ulimit -c unlimited
```

See [8] for the next steps finding the location of the core dump in your code.

6.4 Comparing the run time for different implementations

You sometimes need to compare the run time and required memory size of functions (algorithms) for the same task because algorithms can differ in implementation but the result should be the same if you use the same input. We want to predict the performance (run time, CPU usage) if the number of input data is increased.

We need to express the run time of functions in terms of the size of the input. The size of the input data is indicated by n. We use the *O notation* or "big-Oh-notation" to classify the *time complexity* of algorithms (run time efficiency). The *O* stands for *order of complexity*, you only consider the highest contribution, equals the highest order of n, to the computation time (upper bound).

Suppose you have analysed two functions and have expressed their run times in terms of n. Function A requires $100n+1$ steps and function B requires n^2+n+1 steps. The leading term is term with the highest exponent. For sufficient large n function A needs $100n$ steps and function B needs n^2 steps. All functions with a leading term n^2 belong to $O(n^2)$ en with a leading term n belongs to $O(n)$.

Some *O notation* examples:

- $O(1)$ The computation time is a constant, independent of the number n of input data. The computation time is always at most a constant value.
- $O(n)$ Linear complexity, the computation time is proportional to the number n of input data. The C standard library function *strlen*() has complexity $O(n)$, n is the size of the input string. A *for* loop that traverses an array is usually linear, as long as all of the operations in the body of the loop have complexity $O(1)$. The SLL function *showSLL*() has $O(n)$ time complexity.
- $O(n^2)$ The computation time is quadratic to the number n of input data. Whenever n is doubled the computation time increases fourfold! A number of sort algorithms (for example bubble sort and insertion sort) has this time complexity.

$O(n^3)$ Cubic computation time. Whenever n is doubled the computation time increases eightfold.

$O(2^n)$ Exponential complexity, whenever n doubled the computation time squares. Functions with this complexity are only useful for small data sizes.

$O(log(n))$ Logarithmic complexity, doubling the number n of input data has little effect on the computation time.

The O notation for functions indicates *scalability* for large n values. A function that is $O(n^2)$ will scale worse than one that is $O(n\ log(n))$, but better than one in $O(n^3)$. You need to know this notation in practice for choosing the appropriate implementations of functionality, as well as extrapolating test performance (small n values) into production performance (large n values in released code).

In time-critical applications you are interested in worst case execution time values, the guarantee that a function will always finish on time.

We can determine the time complexity of a function based on the type of statements used in this function, see [11].

Code example:

```
1  void clearSLL(node_t **ppHead)  /* n is the number of nodes */
2  {
3     node_t *pToBeRemoved = *ppHead;        /*   c1  O(1)  */
4     node_t *pNext = NULL;                  /*   c2  O(1)  */
5
6     while (pToBeRemoved != NULL)           /* n*c3  O(n)  */
7     {
8        pNext = pToBeRemoved->pNextNode;    /* n*c4  O(n) */
9        free(pToBeRemoved);                 /* n*c5  O(n) */
10       pToBeRemoved = pNext;               /* n*c6  O(n) */
11    }
12    *ppHead = NULL;                        /*   c7  O(1) */
13 }
```

Total execution time:

$$c1 + c2 + n * c3 + n * c4 + n * c5 + n * c6 + c7$$

$$c1 + c2 + c7 + (c3 + c4 + c5 + c6) * n$$

Highest order in the expression (is dominant) for the total execution time is:

$$(c3 + c4 + c5 + c6) * n$$

The function *clearSLL*() has time complexity $O(n)$.

Compilers can be configured to produce highly run-time efficient code (highest code performance). This kind of code can not be debugged any more, because the relation between the C code lines and the optimised compiled code has been lost. Reducing code size might reduce performance!

6.5 Exercises

Exercise1

Implement the API function *sizeSLL*() for determine the size of an SLL (the SLL size equals the number of nodes). Prototype:

```
size_t  sizeSLL(const node_t *pHead);
```

Why should you use *const*?

The function *sizeSLL*() is very similar to *showSLL*(), instead of showing the data contents of a node it should increment the value of a local variable representing the size of the SLL. Draw a diagram for showing the counting of nodes in an SLL containing 3 nodes.

What happens to the execution time of the function if you double the number of nodes in the SLL? What is the *time complexity* of this function in *big O* notation?

Exercise2

Consider a struct containing a pointer to dynamic allocated memory. Structs can be assigned to each other for copying. What kinds of problems can occur when you copy such a struct to another one?

Draw diagrams with structs and pointers showing the runtime problem. Run the next program (in a Linux development environment) and do a memory check by running the next command:

```
valgrind ./deepcopy
```

Listing 6.6: deepcopy/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct {
6     int count;
7     char *pText; /* must point to dynamic allocated memory! */
8  } problem_t;
9
10 int deepCopy(problem_t *pDestination, const problem_t *pSource);
11
12 int main(void)
13 {
14    problem_t s1 = {1, NULL}; /* do not do ... = {1, "ABC"} */
15    problem_t s2 = {2, NULL};
16
17    s1.pText = (char *)malloc(4 * sizeof(char)); /* array of 4 chars */
18    if (s1.pText != NULL)
19    {
20       strcpy(s1.pText, "111");
21       s2 = s1;           /* shallow copy, problem? */
22       s2.pText[0] = 'A'; /* problem? */
23
24       printf("s1 = %d %s\n", s1.count, s1.pText);
25       printf("s2 = %d %s\n", s2.count, s2.pText);
26
27       free(s1.pText); /* problem? */
28    }
29    s2.pText[1] = 'B'; /* problem? */
30
31    printf("s1 = %d %s\n", s1.count, s1.pText);
32    printf("s2 = %d %s\n", s2.count, s2.pText);
33
34    return 0;
35 }
36
37 int deepCopy(problem_t *pDestination, const problem_t *pSource)
38 {
39    /* TODO implementation, update main() code,
40     * call this function in main() to avoid the problems of
41     * a shallow copy.
42     */
43
44    /* TODO should return an error code if memory allocation fails */
45    return 0;
46 }
```

Exercise3

Consider the next two implementations for determining if an SLL is empty.

First implementation:

```
1 bool isEmptySLL(const node_t *pHead)
2 {
3    return sizeSLL(pHead) == 0;
4 }
```

Second implementation:

```
1 bool isEmptySLL(const node_t *pHead)
2 {
3    return pHead == NULL;
4 }
```

Compare the run-time complexity of these two implementations. Which one is preferable?

Exercise4

Program a function for "deep" copying *problem_t* structs containing an *int* and a pointer *char** to dynamic allocated memory, and the contents of the dynamic allocated *char* array!

This function must solve the problems of a shallow copy causing an ownership problem because after the copy you will have two pointers pointing to the same allocated data in memory. If one of the pointers will deallocate the memory the second pointer becomes a *dangling pointer* (points to de-allocated memory).

Prototype:

*int deepCopy(problem_t * pDestination, const problem_t * pSource)*

The struct where *pDestination* is pointing to, must allocate its own external array of characters for avoiding an ownership problem.

Programming steps for implementing the function *deepCopy()* in *pseudo code* (a step-by-step written outline of your code that you can gradually transcribe into the C programming language):

- free the allocated memory pointed by the pointer in **pDestination* (for preventing memory leakage).
- do a shallow copy: **pDestination = *pSource;*

- determine the size of the allocated memory pointed by the string pointer in **pSource* (string length + 1).
- allocate dynamic memory with the determined size and overwrite the string pointer in **pDestination* with the value returned by *malloc*().
- copy the string pointed by the string pointer in **pSource* to the location pointed to by the string pointer in **pDestination*.

The returned *int* of *deepCopy*() should be an error value (not equal to 0) if the dynamic memory allocation fails.

Test this function in *main*() by replacing the shallow copy *s2 = s1;* by a deep copy calling the function *deepCopy*(&*s*2, &*s*1);.

Use in a Linux development environment *valgrind* for checking memory allocation problems.

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Linus Torvalds

7 Queues

7.1 Introduction

A *queue* is a First-In-First-Out (FIFO) data structure for storing data items in nodes. FIFO: the first data item inserted into a queue is the first data item to leave. Middle data items are logical not accessible.

Queue operations [10]:

- queue insertion (pushing or enqueuing) is done at the back (the rear) of the queue.
- deleting (popping or dequeuing) is done at the front of the queue.

A queue is used for buffering data between a data producer and a data consumer. Producer and consumer can work at different speed, data is transferred asynchronously, a buffer (queue) is necessary. A data producer and consumer can be different programs exchanging data controlled by the operating system (like Linux, Embedded Linux, MacOS, Windows10, VxWorks, Android or FreeRTOS) functions.

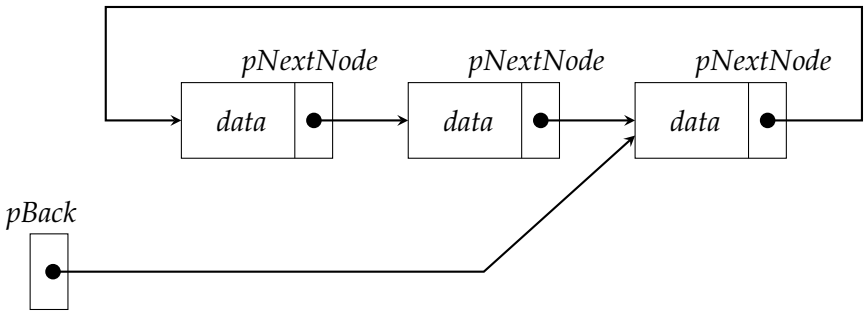
7.2 Implementation strategies for queues

Queues can be implemented in several ways. Each way of implementing a queue will influence the code size and the performance of the API functions.

Commonly used implementations are:

- Static linear array, fixed sized buffer (in a C array and two indices one for the front and one for the back, no run-time overhead caused by dynamic allocation and deallocation of memory).
- Static circular array, fixed sized circular buffer (in a C array and two indices one for the front and one for the back, updated modulo the size of the array).
- Singly Linked List (SLL) , dynamic sized buffer. Every node has a single pointer to another node. To keep track of the queue, you need two pointers to the sequence of nodes, one to the back and one to the front of the queue.
- Circular Singly Linked List (CSLL) , dynamic sized buffer. In a circular list, the back node will point to the front node in the sequence of nodes. So *pFront* equals *pBack->pNextNode*. A CSLL needs only one pointer (queue owner) to the back of the queue *pBack* for managing its contents. If a queue is empty *pBack* must equal NULL.

A three node CSLL:



7.3 Queue CSLL implementation

The SLL and CSLL implementations must use dynamic memory allocation because the code must be able to add and remove nodes at runtime. You must always first determine the interface functions (API) before you can implement them. The set of interface functions should not reflect implementation details of the queue (abstraction). If you should change the CSLL implementation, the calling program should need no updates (API remains te same).

The module *Queue* is implemented in a modular way in two files: *queue.h* and *queue.c*. The C API for de module Queue is based on the C++ standard queue class API. The member class function names are used for the C function names. Every C function name is suffixed by *Queue* because in C every function is global and the names must be unique.

Not all the queue API functions are implemented, they are left as an exercise for the reader, see 7.5. The code is compilable because these not completed functions have empty function bodies or contain only a return statement with an appropriate constant value.

Some comments are documentation type comments in *Doxygen* style. *Dogygen* is able to read these comments in the code files and generate hyper-linked API documentation in HTML format [9]. *Doxygen* is a documentation system not only for C but also for C++, Java, Python, VHDL and PHP code.

Listing 7.1: queueCSLL/app/queue.h

```
1  #ifndef QUEUE_H
2  #define QUEUE_H
3
4  #include <stdlib.h>
5
6  #define TEXT_SIZE 20 /*!< Limits the size of the string data in node. */
7
8  /*!
9   * This struct must not contain pointers pointing outside the struct.
10  */
11 typedef struct {
12    int intVal;
13    char text[TEXT_SIZE];
14 } data_t;
15
16 /*!
17  * Node for the circular linear linked list (singly linked list).
18  */
19 typedef struct node {
20    data_t data;
21    struct node *pNextNode;
22 } node_t;
23
24 /*!
25  * Queue: First In First Out (FIFO).
26  * Implementation: Circular Singly Linked List (CSLL).
27  *
28  * A queue has a front (for popping, is removing data from the queue) and
29  * a back (for pushing, is inserting data into the queue).
30  *
31  * Needs only one external pointer pBack, that points at the back node.
32  * The back node contains a pointer pointing to the front of the queue
33  * (circular data structure).
34  */
35 typedef struct {
36    node_t *pBack;
37    /*! node_t *pFront; only necessary for SLL implementation. */
38 } queue_t;
39
40 /*!
41  * Creates queue with one node containing data.
42  */
43 void createQueue(queue_t *pQueue, data_t data);
```

```
44 /*!
45  * Checks is queue is empty.
46  * \return 1 if empty else 0
47  */
48 int emptyQueue(const queue_t *pQueue);
49 /*!
50  * \return Number of nodes in the queue.
51  * \todo Add implementation.
52  */
53 size_t sizeQueue(const queue_t *pQueue);
54 /*!
55  * \return pointer to the front node data of the queue.
56  */
57 data_t *frontQueue(const queue_t *pQueue);
58 /*!
59  * \return pointer to the front node data of the queue.
60  */
61 data_t *backQueue(const queue_t *pQueue);
62 /*!
63  * Push new data to the back of the queue.
64  * \pre pQueue != NULL
65  */
66 void pushQueue(queue_t *pQueue, data_t data);
67 /*!
68  * Remove data from the front of the queue.
69  * \pre pQueue != NULL, size >= 1
70  */
71 void popQueue(queue_t *pQueue);
72 /*!
73  * Emptying (delete, deallocate) queue.
74  * \todo Add implementation.
75  */
76 void deleteQueue(queue_t *pQueue);
77 /*!
78  * Shows the contents of queue in text, node by node.
79  */
80 void showQueue(const queue_t *pQueue);
81
82 #endif
```

Listing 7.2: queueCSLL/app/queue.c

```
1  #include "queue.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  void createQueue(queue_t *pQueue, data_t data)
7  {
8     node_t *pNew = (node_t *)malloc(sizeof(node_t));
9     pQueue->pBack = pNew;
10    if (pNew != NULL)
11    {
12       pNew->data = data; /* copy input struct data */
```

```
13          pNew->pNextNode = pNew;
14       }
15    }
16
17    int emptyQueue(const queue_t *pQueue)
18    {
19       return pQueue->pBack == NULL;
20    }
21
22    size_t sizeQueue(const queue_t *pQueue)
23    {
24       size_t size = 0;
25       /* local pointer for traversing all nodes in queue */
26       const node_t *pSize = pQueue->pBack;
27
28       return size;
29    }
30
31    data_t *frontQueue(const queue_t *pQueue)
32    {
33       data_t *pFrontData = NULL;
34       if (!emptyQueue(pQueue))
35       {
36          pFrontData = &(pQueue->pBack->pNextNode->data);
37       }
38       return pFrontData;
39    }
40
41    data_t *backQueue(const queue_t *pQueue)
42    {
43       data_t *pBackData = NULL;
44       if (!emptyQueue(pQueue))
45       {
46          pBackData = &(pQueue->pBack->data);
47       }
48       return pBackData;
49    }
50
51    void pushQueue(queue_t *pQueue, data_t data)
52    {
53       node_t *pNew = (node_t *)malloc(sizeof(node_t));
54       if (pNew != NULL)
55       {
56          pNew->data = data;
57          pNew->pNextNode = pQueue->pBack->pNextNode;
58          pQueue->pBack->pNextNode = pNew;
59          pQueue->pBack = pNew;
60       }
61    }
62
63    void popQueue(queue_t *pQueue)
64    {
65       if (pQueue->pBack != NULL)
66       {
67          node_t *pDelete = pQueue->pBack->pNextNode;
68          if (pDelete == pQueue->pBack)
69          {
70             /* size queue == 1 */
71             pQueue->pBack = NULL;
72          }
```

```
73        else
74        {
75           pQueue->pBack->pNextNode = pDelete->pNextNode;
76        }
77        free(pDelete);
78     }
79  }
80
81  void deleteQueue(queue_t *pQueue)
82  {
83     /* local pointer for traversing all nodes in queue */
84     node_t *pDelete = pQueue->pBack;
85  }
86
87  void showQueue(const queue_t *pQueue)
88  {
89     const node_t *pNext = pQueue->pBack;
90
91     if (pNext == NULL)
92     {
93        printf("Queue is empty\n");
94     }
95     else
96     {
97        printf("Queue contains:\n");
98        do
99        {
100          pNext = pNext->pNextNode;
101          printf(
102             "pNode = %p  Data = '%d' '%s'\n"
103             "                       pNextNode = %p\n",
104             pNext, pNext->data.intVal, pNext->data.text, pNext->pNextNode);
105       } while (pNext != pQueue->pBack);
106    }
107 }
```

The next application in *main*() shows how to use some queue API functions. The output shows in several steps the resulting queue contents.

Listing 7.3: queueCSLL/app/main.c

```
1  #include "queue.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void)
7  {
8     data_t data = {1, "Hello queue"};
9     /* Create empty queue, pBack = NULL */
10    queue_t queue = {NULL};
11
```

```
12      showQueue(&queue);
13      printf("\nCreate queue\n");
14      createQueue(&queue, data);
15      showQueue(&queue);
16
17      data.intVal++;
18      printf("\nAdd new data to queue\n");
19      pushQueue(&queue, data);
20      showQueue(&queue);
21
22      data.intVal++;
23      printf("\nAdd new data to queue\n");
24      pushQueue(&queue, data);
25      showQueue(&queue);
26
27      if (!emptyQueue(&queue))
28      {
29         printf("\nFront iValue: %d\n", frontQueue(&queue)->intVal);
30         printf("Front text:   %s\n", frontQueue(&queue)->text);
31         printf("Back  iValue: %d\n", backQueue(&queue)->intVal);
32         printf("Back  text:   %s\n", backQueue(&queue)->text);
33      }
34
35      printf("\nPop queue\n");
36      popQueue(&queue);
37      showQueue(&queue);
38
39      printf("\nPop queue\n");
40      popQueue(&queue);
41      showQueue(&queue);
42
43      printf("\nPop queue\n");
44      popQueue(&queue);
45      showQueue(&queue);
46
47      return 0;
48  }
```

Output:

```
Queue is empty

Create queue
Queue contains:
pNode = 0x564e67cb1670  Data = '1' 'Hello queue'
                        pNextNode = 0x564e67cb1670

Add new data to queue
Queue contains:
pNode = 0x564e67cb1670  Data = '1' 'Hello queue'
                        pNextNode = 0x564e67cb16a0
pNode = 0x564e67cb16a0  Data = '2' 'Hello queue'
                        pNextNode = 0x564e67cb1670

Add new data to queue
Queue contains:
pNode = 0x564e67cb1670  Data = '1' 'Hello queue'
                        pNextNode = 0x564e67cb16a0
pNode = 0x564e67cb16a0  Data = '2' 'Hello queue'
                        pNextNode = 0x564e67cb16d0
pNode = 0x564e67cb16d0  Data = '3' 'Hello queue'
                        pNextNode = 0x564e67cb1670

Front iValue: 1
Front text:   Hello queue
Back  iValue: 3
Back  text:   Hello queue

Pop queue
Queue contains:
pNode = 0x564e67cb16a0  Data = '2' 'Hello queue'
                        pNextNode = 0x564e67cb16d0
pNode = 0x564e67cb16d0  Data = '3' 'Hello queue'
                        pNextNode = 0x564e67cb16a0

Pop queue
Queue contains:
pNode = 0x564e67cb16d0  Data = '3' 'Hello queue'
                        pNextNode = 0x564e67cb16d0

Pop queue
Queue is empty

Pop queue
Queue is empty
```

7.4 Pushing data into a queue

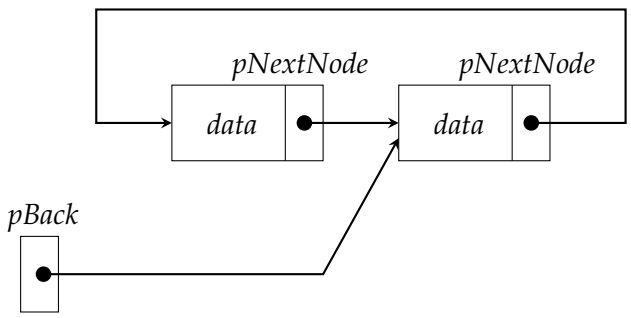
To understand the dynamics of a queue API function implementation, draw the nodes and the pointers in diagrams. Drawing diagrams is an absolutely necessary design step. The next code block shows the implementation of the *pushQueue*() function. The first parameter is the pointer *pQueue* pointing to the struct containing the *pBack* pointer. The second parameter is passed by value, the input data struct *data* is copied.

The implementation of te function is based on several design diagrams showing some intermediate steps for adding one node to the queue.

Listing 7.4: queueCSLL/app/queue.c (line 51..61)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
7        pNew->pNextNode = pQueue->pBack->pNextNode;
8        pQueue->pBack->pNextNode = pNew;
9        pQueue->pBack = pNew;
10    }
11 }
```

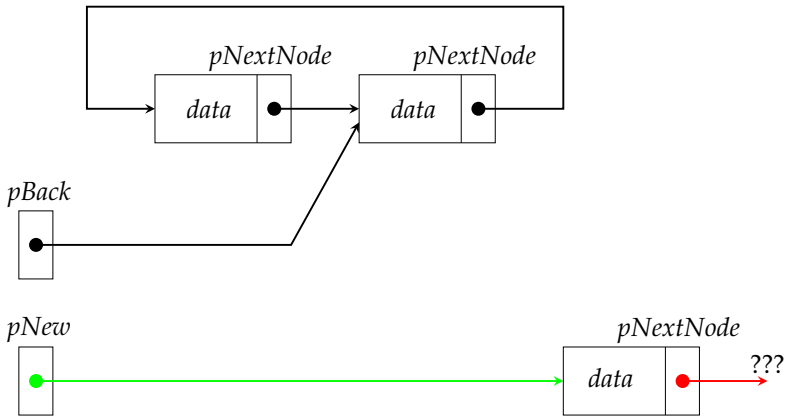
Starting-point: a queue contains two nodes. Every node contains some data and a pointer to the next node. The pointer *pBack* is the owner of the queue, pointing to the back of the queue and is implemented in a *queue_t* struct. This struct containing *pBack* is not shown in the following diagrams, only *pBack* is shown. In the code this is *pQueue->pBack*. The front of the queue is reachable by *pQueue->pBack->pNextNode*.



Listing 7.5: queueCSLL/app/queue.c (line 51..56)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
```

- line 3 dynamically creates a new node.
- line 4 checks if the allocation has succeeded. The pointer *pNew* is pointing to this newly allocated node. The *pNextNode* of this node is not yet defined. This a *dangling pointer*. A dangling pointer contains an undefined memory address and should never be dereferenced because the result will cause undefined behaviour.
- line 6 copies the input struct *data* by an assignment, because structs can be copied by an assignment (copy by value).

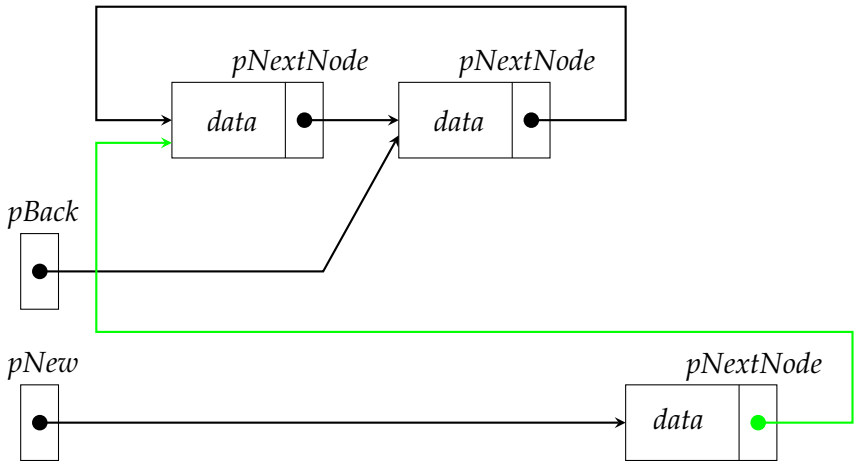


Solving the dangling pointer problem (see the three question-marks ??? in the above diagram) is the next step in the function *pushQueue*().

Listing 7.6: queueCSLL/app/queue.c (lines 51..57)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
7        pNew->pNextNode = pQueue->pBack->pNextNode;
```

- line 7 connects the new node to the CSLL under construction, solving the dangling pointer problem.

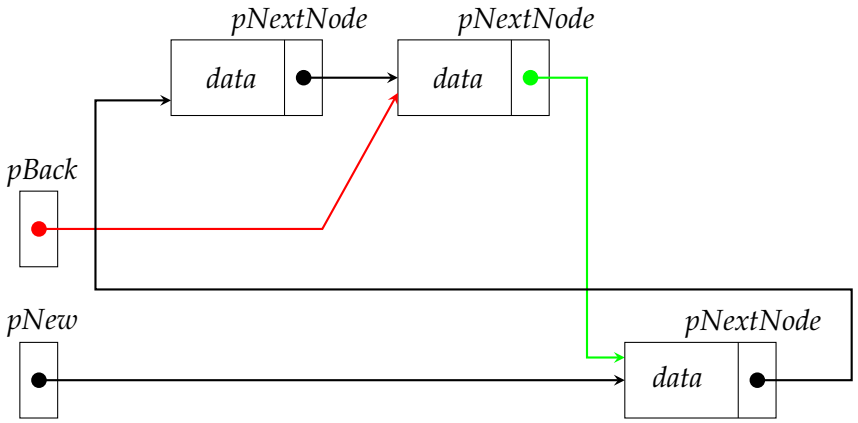


The pointer *pBack* needs to be updated

Listing 7.7: queueCSLL/app/queue.c (lines 51..58)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
7        pNew->pNextNode = pQueue->pBack->pNextNode;
8        pQueue->pBack->pNextNode = pNew;
```

- line 8, *pBack* is not yet pointing to the back of the queue under construction. This must be solved in a next step.



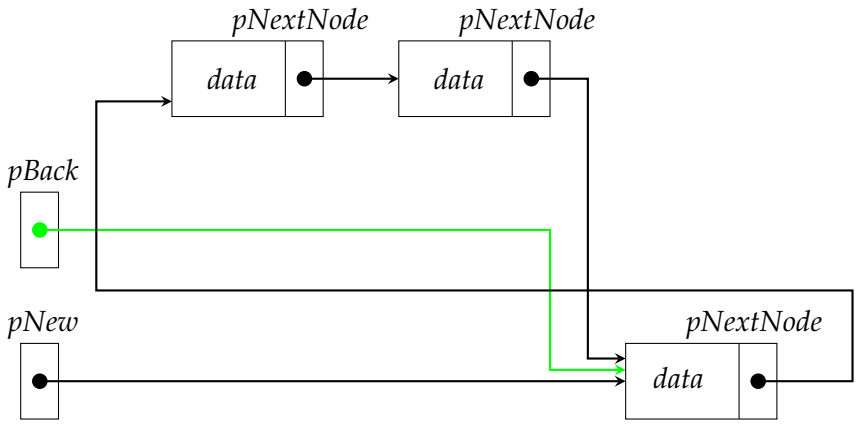
Updating the *pBack* pointer.

Listing 7.8: queueCSLL/app/queue.c (lines 51..59)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
7        pNew->pNextNode = pQueue->pBack->pNextNode;
8        pQueue->pBack->pNextNode = pNew;
9        pQueue->pBack = pNew;
```

- line 9, *pBack* is now pointing to new back of the queue: the newly added data node.

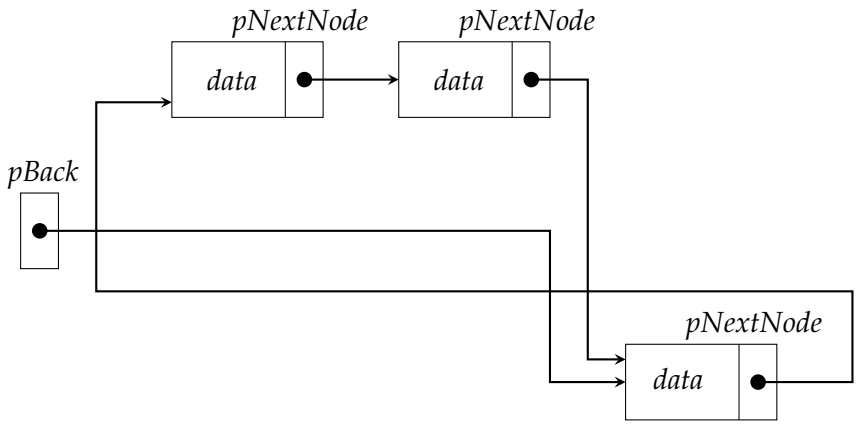
The addition of a new node to the queue is shown step by step. The final result will be shown in the next diagrams.



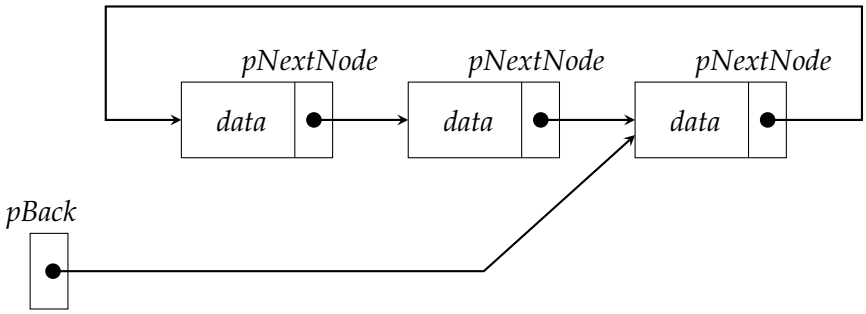
Listing 7.9: queueCSLL/app/queue.c (lines 51..61)

```
1  void pushQueue(queue_t *pQueue, data_t data)
2  {
3     node_t *pNew = (node_t *)malloc(sizeof(node_t));
4     if (pNew != NULL)
5     {
6        pNew->data = data;
7        pNew->pNextNode = pQueue->pBack->pNextNode;
8        pQueue->pBack->pNextNode = pNew;
9        pQueue->pBack = pNew;
10    }
11 }
```

- line 11, end of function body, local pointer variable *pNew* is removed from the *stack* (special region in memory for storing local variables by each function call).



The final result is a queue containing three nodes after calling *pushQueue*() in a clear diagram in the same style as the drawing of the initial queue containing two data nodes:



7.5 Exercises

Exercises to the CSLL queue implementation.

Exercise1

Draw a diagram for an empty CSLL and a CSLL containing one node.

Exercise2

Implement the queue API function: *sizeQueue*() for determining the number of nodes in a queue. Before implementing this function draw a diagram for visualising the steps of counting the number of nodes. Start with a queue containing 3 nodes and the necessary pointers. Test this function in *main*().

What is the *time complexity* of this function in *big O* notation?

Exercise3

Implement the queue API function: *deleteQueue*() for deleting all nodes in a queue. Before implementing this function draw several diagrams for visualising the steps for deleting all nodes. Start with a queue containing 3 nodes and the necessary pointers. Test this function in *main*().

What is the *time complexity* of this function in *big O* notation?

Exercise4

The queue API function: *createQueue*() is not fully implemented. It lacks checking if the pointer *pBack* is already pointing to an existing queue. This queue must be deleted otherwise this will cause a leaking memory problem. Why? Test this function in *main*().

Exercise5

The following queue API functions *createQueue*() and *pushQueue*() return no error code indicating if the allocation of memory has succeeded or not. Which type and values should be used for the error code? Implement this feature. Why is this important? Are these functions easy to test in *main*()? Why?

Writing in C or C++ is like running a chain saw with all the safety guards removed.

Bob Gray

8

Compare floating-point values

8.1 Introduction

Comparing of floating-point numbers needs special attention. You must never compare *float* and *double* type values by the == operator .

Comparing *integral types*, for example *long*, *int* and *char* values, by the == operator is correct. You can not use the == operator for comparing strings. Comparing two null-terminated strings can be done by the standard *strcmp*() function. This function compares the strings char by char until the corresponding chars are different or the null character is reached.

8.2 Comparing floating-point values

The next program shows the problems of the comparison of floating-point values by ==. The problems are solved by calling the *fequal()* function. This function is not available in the standard C library. The developer is responsible for choosing an appropriate small positive value for the EPSILON constant determined by the application domain.

The next code example defines EPSILON in a *#define* macro using the C scientific notation *1e-8* equals 10^{-8} for this small positive value.

Listing 8.1: compare/main.c

```
1  #include <float.h>
2  #include <math.h>
3  #include <stdio.h>
4
5  #define EPSILON 1e-9
6
7  int fequal(double a, double b);
8
9  int main(void)
10 {
11    double d1 = 0.1;
12    double d2 = 0.2;
13    float f1 = 0.1;
14
15    printf(
16       "Smallest double value != 0.0\n"
17       "        = %.17lf\n\n",
18       DBL_EPSILON);
19    printf(
20       "Smallest float value != 0.0\n"
21       "        = %.17f\n\n",
22       FLT_EPSILON);
23
24    printf("d1      = %.17lf\n", d1);
25    printf("f1      = %.17f\n", f1);
26    printf("d2      = %.17lf\n\n", d2);
27    printf("d1 + d2 = %.17lf\n\n", d1 + d2);
28
29    if (d1 == f1)
30    {
31       printf("double d1 0.1 == float f1 0.1\n\n");
32    }
33    else
34    {
35       printf("double d1 0.1 != float f1 0.1\n\n");
36    }
37
38    if (d1 + d2 == 0.3)
39    {
40       printf("d1 + d2 == 0.3   0.1 + 0.2 == 0.3\n\n");
41    }
42    else
43    {
44       printf("d1 + d2 != 0.3   0.1 + 0.2 != 0.3\n\n");
45    }
46
47    if (fequal(d1 + d2, 0.3))
48    {
49       printf(
50          "fequal(d1 + d2, 0.3)   0.1 + 0.2 == 0.3\n"
51          "                       for EPSILON == "
52          "%.15lf\n\n",
```

```
53            EPSILON);
54      }
55      else
56      {
57         printf(
58            "fequal(d1 + d2, 0.3)   0.1 + 0.2 != 0.3\n"
59            "                       for EPSILON == "
60            "%.15lf\n\n",
61            EPSILON);
62      }
63
64      return 0;
65   }
66
67   int fequal(double a, double b)
68   {
69      return fabs(a - b) < EPSILON;
70   }
```

The output of the program for the comparing of doubles and floats using the == operator, shows the differences between floats and doubles, and the difference between the programmed initial values in text format (0.1 and 0.2) and the final representation of these values in binary format in the variables. The computer can not store the literal text values 0.1 and 0.2 in memory.

The next output shows the slightly different values generated by the compiler in memory:

```
Smallest double value != 0.0
        = 0.00000000000000022

Smallest float value != 0.0
        = 0.00000011920928955

d1      = 0.10000000000000001
f1      = 0.10000000149011612
d2      = 0.20000000000000001

d1 + d2 = 0.30000000000000004

double d1 0.1 != float f1 0.1

d1 + d2 != 0.3   0.1 + 0.2 != 0.3

fequal(d1 + d2, 0.3)   0.1 + 0.2 == 0.3
                       for EPSILON == 0.000000001000000
```

The output examples show that binary representations can be close enough to the literal values for practical purposes.

8.3 Binary fractions

It is often not possible to represent a simple fraction exactly in a binary format (limited number of bytes). Binary fractions can only be represented by the summation of 2^{-1} equals $\frac{1}{2}$, 2^{-2} equals $\frac{1}{4}$, 2^{-3} equals $\frac{1}{8}$, 2^{-4} equals $\frac{1}{16}$, etc.

Most decimals have infinite representations in binary!

For example one-tenth, or 0.1 (decimal) has an infinite binary representation, containing the repeating pattern 0011:

```
0.0001100110011001100110011001100110011001100110011...
```

Only a limited number of bits in *double* and *float* typed variables can be used in real systems. See *d*1 and *f1* in the output of the code example.

8.4 Exercises

Exercise1

Why should you never use floating-point variables as a loop counter in a for-loop? Code example for-loop:

```
1 for (float x = 0.1f; x <=1.0f; x += 0.1f)
2 {
3    /* do something, use x */
4 }
```

How many times do you expect that this loop will be iterated? Why is the next code block preferred?

```
1 for (int i = 1; i <= 10; i++)
2 {
3    float x = i * 0.1;
4    /* do something, use x */
5 }
```

See 'Rule 05. Floating Point' in [15].

Exercise2

How can you compare two structs?

It is not enough to do your best: you must know what to do, and then do your best.

Edwards Deming

9

Software engineering, debugging and testing

9.1 Introduction

This chapter contains a very brief concise introduction to software testing.

"A software *bug* is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result or to behave in unintended ways [2]."

Second improved definition by Yegor Bugayenko: "A software *bug* is an error, flaw, failure, or fault in a computer program or system that causes it to violate at least one of its functional or non-functional requirements [3]."

"Testing is the key method used for dynamic verification of a program or system. It does this by running a discrete set of test cases. The test cases are suitably selected from a finite, but very large, input domain. During testing the actual behaviour is compared with the intended or expected behaviour [1]."

A system controlled by software should show predictable consistent behaviour.

9.2 Important debugging and testing guidelines

Some important guidelines:

- Fix bugs as soon a possible in the code you are responsible for. Avoid the frustration to familiarise yourself again with a piece of code you once knew.
- Unfixed bugs can camouflage other bugs and can annoy the entire development team [4].
- The earlier a bug is fixed, the less expensive it is to fix.
- Regression testing: it is not enough to pass a test once. After any modification of the software or hardware of a system, it should be retested.
- Professional ethics: software engineers should always ensure testing, debugging and review of their code sufficiently before shipping.

9.3 Testing and testability

Testing is part of each different development phase for example in the V-Model (also known as Verification and Validation model) for system development:

- acceptance testing
- system testing
- integration testing
- unit testing

The V-model emphasises requirements-driven design and testing. These test activities start in parallel with the development activities. Tests are necessary because in almost all cases developers can not prove mathematically if a (complex) program is correct. Correct means: "does what it is supposed to do". Software testing involves the execution of a program with the intent of finding *software bugs* (errors or other defects) [13].

In system development projects you not only test for correctness but also for example:

- *performance*, a program must meet its performance requirements (non-functional requirements).
- *robustness*, a program must detect invalid unexpected input to avoid undefined behaviour or even system crashes ("garbage in, is garbage out").

- *security*, to check if a system is vulnerable to malicious attacks, if anyone is able to hack the system.
- *usability*, to evaluate the interaction of a system user interface with representative users to improve the user experience.
- *acceptance testing*, performed by the end-user, to check if the system conforms to the user specifications (requirements).

The *testability* of developed code can be improved by using some important software engineering guidelines for managing the *code complexity* :

- Limit the size of functions. Avoid a large amount of complex logic in one function. Every function should be as small as possible and have one main responsibility.
- Limit the size of complicated single line expressions. Use multiple code lines and variables for caching intermediate results.
- Use a modular approach for developing an application and separate as much as possible the user interface code from the core code of the application.

9.4 Test strategies: white, black and grey box testing

Types of code correctness testing:

- *White box testing* or *logic-driven testing*, the internals are fully known. Testing as a developer. Also known as clear box testing, structural testing and code based testing.
- *Black box testing*, testers are not known to the internals. Testing as an (end) user. Also known as closed box testing and functional testing.
- *Grey box testing*, some internals relevant for testing are known. Testing as a user with access to (some) internals. In this approach you need to suspect the weak points in the code: error guessing (based on intuitive and expert knowledge).

A *unit test* is a white box test done by the developer of the code. Unit tests are based on test cases and implemented in code using a test framework. A lot of unit test frameworks are available, not only for C but for all major programming languages.

The test code is separated from the code of the application under development. We can run the unit test code every time when a tested function is changed (refactored) by a developer. *Refactoring* is changing existing code without changing its external behaviour.

In large software projects, a program can be fully built and unit tested on a build server (automated tests) at night (nightly build). The test results are then shown in the morning to all developers before starting their work.

Try to make each unit test independent of all the others.

After testing all units you must do *integration testing* . Individual units are combined and tested as a group. A *system test* tests a completely integrated system to verify that the system meets its requirements.

9.4.1 White box testing

You must examine the implementation of functions in detail. Because you know the internals of a function you are able to design the test cases. Because a function can contain one or more if-else statements, you must specify several test cases to follow all the execution paths in the function.

In if-else statements you can find complex conditions for deciding which execution path to follow. For that reason, you must design test cases that all parts of a complex condition become *true* and $false$.

For example a tested function contains the next if-statement:

$$if\ (a\ \&\&\ b\ ||\ c)$$

you have 8 possible combinations (for the variables a, b and c) of *true* and $false$ values. It is necessary to consider different input values for the function under test to enable all possible logical combinations in the if-statement.

If a function is too large (too complex), these kind of functions are difficult or even impossible to test.

If a function returns error codes you also must design test cases for every error code value that can be returned.

9.4.2 Black box testing

Black box testing is also known as data-driven or input/output-driven testing. Because you do not know the internals of a function, you are only able to design the test cases based on input parameter values. The test data is derived solely from the specifications. To use all possible input value combinations is impractical, if not impossible. Because exhaustive testing takes to much time.

9.4.3 Grey box testing

Grey box testing is a combination of white-box and black-box testing. You do not use all the details of the implementation. This is necessary if not all the details are understood or it takes to much time to find out.

9.5 Boundary value analysis

Boundary value analysis: always take for input values in test cases also the boundaries of the input range of values (input space). Because many bugs are related to boundary value conditions.

For example if strings are input for testing also use an empty string (only containing a terminating null character) or a string containing one character. If a floating-point input ranges from -1.0 to 1.0 (input space), write test-cases for the next input values -1.0, -0.999, 0.999 and 1.0.

It ia also important to select input values that should result in boundary values for the result space.

Always consider to use boundary value analysis in all kind of test strategies. If practised correctly, it is one of the most useful test-case design approaches.

9.6 Manual testing

Different ways of manual testing (not automated):

- *Desk checking* is a manual technique for checking the correctness of code (white box testing), just by reading the code line by line and using paper and pencil for tracking the values of each variable and the results of calculation and logic. *Desk checking* will solve simple (non-severe) defects and are only effective if taken seriously. This informal technique will speed up other more formal code inspections.
- In a *walkthrough* a developer will describe its code to other developers and ask for comments. This can solve some errors and can improve the quality of the code. The participants can desk check the code in advance. A *walkthrough* can also be used for improving the coding skills of junior developers.
- In a *code review* is the examination of code in a team ideally led by a moderator, who is not the author of the code. Reviewers prepare a review report with a list of findings for the review meeting. This kind of review is usually performed

as a *peer review* without management participation. Development teams could use code reviews to mentor junior developers by senior developers pointing out best practices and showing how to solve (complex) bugs.

9.7 Test driven development

Test driven development (TDD) is a method which encourages the developers to implement sufficient unit tests prior to the writing of the actual production code. The unit tests create a detailed specification of the functions under development. Unit tests can be considered as the documentation of a function. This means that you first need to focus on the development of the function interface not on the implementation.

The body of the first version of the functions is empty or returns some constant value. This is necessary to be able to compile these functions under construction. This will at first result in the failing of all unit tests. The step by step implementation of functions is strongly guided by the unit tests. Because every unit test gives immediate feedback about the correctness of the function code you can significantly reduce the time for reviews and rework.

TDD leads to more modularized, flexible and extensible code. Because a unit test approach requires developers to think in terms of small independently testable functions or other code units.

9.8 Test plan

A *test plan* describes the type of tests, scope of tests, what to test (test cases), tests prioritizing, how to test, pass and fail criteria, problem reporting, when to test, test documentation text templates and who will do the testing.

In large projects a test manager is responsible for creating such a plan. The details of test plans are not in the scope of this book.

9.9 Bad C coding practices

Run time errors (bugs) can cause problems at different severity levels ranging from users can get annoyed, undefined behaviour and crashes of programs, causing high costs or even the death of humans. Bad coding practices will considerably increase the number of bugs.

Some bad C coding practices:

- ignoring compiler warnings or using a low warning level
- not compiling at the highest warning level
- using uninitialised variables in expressions
- using mixed types (signed and unsigned) in arithmetic expressions calls
- not using function prototypes (compiler can not check correct function calls)
- not using *void* for empty parameter lists (compiler can not check correct function calls)
- using ”magic numbers”
- avoiding the use of *const*
- using absolute file paths
- failing to layout the code in a standardised consistent and readable way
- failing to modularize code for managing complexity
- not using code version control (for example Git and Apache Subversion)
- not testing early and often
- reinventing the wheel, writing already available and tested functions

9.10 Reviewing C code for finding common errors

A lot of bugs can be solved by just carefully reading (reviewing) the code function by function, line by line. The next list shows the most common errors in C code:

- using = for comparing, in stead of ==
- comparing floating point values by ==
- off by one error, starting a loop at 1 instead of 0, writing <= instead of < or writing >= instead of >
- negative array indexes
- buffer overflow (overstepping array boundaries)
- division by 0
- integer division in stead of double division

- overflow in calculations
- incrementing and decrementing variables beyond its limits
- leaving characters in the input buffer
- incorrect memory management causing memory leaks
- NULL pointer and dangling pointer dereferencing
- mixed-type expressions

The mentioned common errors are mostly not solved by running the code under test. Because if a (complex) program crashes or shows undefined behaviour you can not easily deduce the location of the error. Applying programming guidelines and reviewing code can solve most of the mentioned common errors. Code reviews actually save time in the long run.

Always run and test your code after small changes. Read carefully the error messages the system showed to you.

The next code example shows mixed type expressions:

Listing 9.1: mixedtypes/main.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     unsigned int uiA = 123;
6     unsigned int uiB = 0;
7     int iC = -567;
8
9     printf("Mixed types in expressions will cause problems!\n\n");
10
11    uiB = uiA + iC;
12    printf(
13       "uiB = uiA + iC;\n"
14       "iC = %d is casted to unsigned int: %u\n"
15       "Mixed type expression causes: uiB = %u\n\n",
16       iC, iC, uiB);
17
18    if (uiA < iC)
19    {
20       printf(
21          "uiA < iC\n"
22          "iC = %d is casted to unsigned int: %u\n"
23          "Mixed type expression causes: 123 < -567 to be true\n\n", iC, iC);
24    }
25    else
26    {
27       printf("OK\n");
```

```
28      }
29
30      return 0;
31  }
```

Output:

```
Mixed types in expressions will cause problems!

uiB = uiA + iC;
iC = -567 is casted to unsigned int: 4294966729
Mixed type expression causes: uiB = 4294966852

uiA < iC
iC = -567 is casted to unsigned int: 4294966729
Mixed type expression causes: 123 < -567 to be true
```

Software testing proves the existence of bugs not their absence.

Anonymous

10 Unit testing

10.1 Introduction

"A unit test is a software development life cycle (SDLC) component in which a comprehensive testing procedure is individually applied to the smallest parts of a software program for fitness or desired operation [16]."

A unit test is implemented in a separate program and contains for every test three steps:

- initialises some input test data
- calls the function under test using the initialised input test data
- compares the results with the expected results

Unit testing finds problems early in the development process. If you can show that isolated units are correct then you can test more easily integrated units (components or subsystems). Unit test code uses test framework libraries and are typically automated tests. If an implementation of a function in a unit is changed, you can re-run the related unit tests: *regression testing*. The next paragraphs will show you how to write test code for automated tests.

This chapter shows some unit test code examples using the *Unity* and the *Catch*2 frameworks.

10.2 Preprocessor macros and *__func__*

A lot of unit test frameworks are available. For C and C++ they often use a lot of pre-processor macro's for the implementation of test-functions. In unit test code you will find a lot of C *functional macros*. The next paragraphs contain code examples for unit testing. In these code examples the pre-processor will expand these functional macros before compiling. The pre-processor does not do any type checking. The compiler will only check the results of the macro expansions.

C99 introduced *__func__* (is not a preprocessor macro), a string containing the name of the current function. The name of an executed function can be used for debugging and logging purposes.

The standard preprocessor directives *__LINE__*, *__FILE__*, *__TIME__* and *__DATE__* can also be used for creating compile time strings intended for diagnostic purposes.

Constant and functional macros can be shared by defining them in header files.

Listing 10.1: macros/main.c

```
1  #include <stdio.h>
2
3  /* Constant macros */
4  #define VALUE 25
5  #define OTHER_VALUE (2 * VALUE)
6
7  /* Function-like macros can take arguments, just like true functions. */
8  /* Macros are not type checked, pre-processor only replaces texts.    */
9  #define SQR(x) ((x) * (x))
10 #define PRINT_VALUE(format, var) printf("%s = %" #format "\n", #var, var)
11
12 /* Multi line macro, use \ */
13 #define TEST(condition)                                                  \
14    if (!(condition))                                                     \
15    fprintf(stderr, "TEST '" #condition "' is not satisfied\n")
16
17 /* C99 variadic macros: macros may have variable number of arguments */
18 #define WARNING(...) fprintf(stderr, "Warning: " __VA_ARGS__)
19
20 int main(void)
21 {
22    int i1 = SQR(5);      /* int i1 = (5) * (5); */
23    int i2 = SQR(i1);     /* int i2 = (i1) * (i1); */
24    int i3 = SQR(1 + i1); /* int i3 = (1 + i1) * (1 + i1); */
25    int i4 = SQR(VALUE);  /* int i4 = (VALUE) * (VALUE); */
26                          /* int i4 = (100) * (100); */
27    double v = SQR(1.1);  /* double v = (1.1) * (1.1); */
28
29    /* Standard predefined macros */
30    printf("C standard version is %ld\n", __STDC_VERSION__);
31    printf("Code compiled: %s %s\n"
```

```
32            "         file: %s\n"
33            "         line: %d in function: '%s'\n\n",
34            __DATE__, __TIME__, __FILE__, __LINE__, __func__);
35
36    PRINT_VALUE(d, SQR(10));
37    PRINT_VALUE(d, OTHER_VALUE);
38    PRINT_VALUE(d, i1);
39    PRINT_VALUE(x, i2);
40    PRINT_VALUE(d, i3);
41    PRINT_VALUE(d, i4);
42    PRINT_VALUE(lf, v);
43    puts("");
44
45    TEST(v > 2.0);
46    TEST(i3 < 1 && i4 < 1);
47    puts("");
48
49    PRINT_VALUE(d, SQR(SQR(3)));
50    WARNING("Macros can become complex, result %d\n", SQR(SQR(3)));
51
52    return 0;
53 }
```

The macro *TEST* is an example of a basic unit test macro showing a message if a condition (logical expression) has failed (equals *false*). The variadic macro *WARNING* shows the power of using a variable number of macro arguments.

Output:

```
C standard version is 199901
Code compiled: Aug  5 2020 19:21:28
         file: main.c
         line: 34 in function: 'main'

SQR(10) = 100
OTHER_VALUE = 50
i1 = 25
i2 = 271
i3 = 676
i4 = 625
v = 1.210000

TEST 'v > 2.0' is not satisfied
TEST 'i3 < 1 && i4 < 1' is not satisfied

SQR(SQR(3)) = 81
Warning: Macros can become complex, result 81
```

The first four output lines show the result of printing the values of predefined macros for *__STDC_VERSION__*, *__DATE__*, *__TIME__*, *__FILE__*, *__LINE__* and *__func__*.

10.3 Unit test C framework Unity

Unity is a straightforward small unit test framework especially for embedded software [17]. This framework contains a lot of utility function macro's for easily creating your own test functions. *Unity* is written in 100% pure C code, it follows ANSI C standards [18]. The default *Unity* version does not support *double* checking. The next code example uses *float*s.

Some *Unity* test macro examples defined in the framework by using *#define* in the file *unity.h*. Just by reading the macro texts you can find out the goal of the macro functions.

The test macro *TEST_ASSERT_FLOAT_WITHIN_MESSAGE* for comparing a *float* to an expected value must contain a float value for determine the within-range (*DELTA* equals 0.001) for the comparison. If a test condition is not satisfied the coded error message text will be printed.

Some examples:

```
TEST_ASSERT(a == 1);

TEST_ASSERT_MESSAGE(a == 2 , "a != 2");

TEST_ASSERT_EQUAL_INT_MESSAGE(5, val, "val != 5");

TEST_ASSERT_EQUAL_INT_ARRAY_MESSAGE(expected, array, 20,
                                    "expected != array");

TEST_ASSERT_FLOAT_WITHIN_MESSAGE(0.001, 5.1, val, "val != 5.1");
```

Example for unit testing the module *checkStrings*, see chapter 3 exercises.

Listing 10.2: checkStrings/test/main.c

```
1  /* Unit tests: using Unity unit test framework */
2
3  #include "checkStrings.h"
4  #include "unity.h"
5  #include <stdio.h>
6
7  const float DELTA = 0.001f;
8
9  void test_trim(void)
10 {
11    printf("---- trim()\n");
```

```
12
13     {
14         const char str[] = "test";
15         const char expected[] = "test";
16         char dest[100] = {'\0'};
17
18         trim(dest, str);
19         TEST_ASSERT_EQUAL_STRING_MESSAGE(expected, dest, "test 1");
20     }
21     {
22         const char str[] = " test test test  ";
23         const char expected[] = "test test test";
24         char dest[100] = {'\0'};
25
26         trim(dest, str);
27         TEST_ASSERT_EQUAL_STRING_MESSAGE(expected, dest, "test 2");
28     }
29     {
30         const char str[] = "\t test \t test test     test\t ";
31         const char expected[] = "test \t test test     test";
32         char dest[100] = {'\0'};
33
34         trim(dest, str);
35         TEST_ASSERT_EQUAL_STRING_MESSAGE(expected, dest, "test 3");
36     }
37 }
38
39 void test_isInteger(void)
40 {
41     printf("\n---- isInteger()\n");
42
43     TEST_ASSERT_TRUE_MESSAGE(isInteger("0"), "test 1");
44     TEST_ASSERT_TRUE_MESSAGE(isInteger("123"), "test 2");
45     TEST_ASSERT_TRUE_MESSAGE(isInteger("-123"), "test 3");
46     TEST_ASSERT_TRUE_MESSAGE(isInteger("+123"), "test 4");
47
48     TEST_ASSERT_FALSE_MESSAGE(isInteger("a123"), "test 5");
49 }
50
51 void test_isMACaddress(void)
52 {
53     printf("\n---- isMACaddress()\n");
54
55     TEST_FAIL_MESSAGE("Tests not yet implemented");
56 }
57
58 int main(void)
59 {
60     UNITY_BEGIN();
61     printf("== Unit tests 'checkStrings' ==\n\n");
62     RUN_TEST(test_trim);
63     RUN_TEST(test_isInteger);
64     RUN_TEST(test_isMACaddress);
65
66     return UNITY_END();
67 }
```

Output for *trim()* and *isInteger()*:

```
== Unit tests 'checkStrings' ==

---- trim()
main.c:62:test_trim:PASS

---- isInteger()
main.c:45:test_isInteger:FAIL: test 3
```

Output for *isMACaddress()*:

```
---- isMACaddress()
main.c:55:test_isMACaddress:FAIL: Tests not yet implemented

-----------------------
3 Tests 2 Failures 0 Ignored
FAIL
```

The next full code project example shows the code unit under test: *functions*. This C module (*functions.h* and *functions.c*) and the unit test code. These functions can be used for time series data processing and analysis. A *time series* is a sequence of data points recorded at regular intervals of time. Review all the unit test code before running the tests. This is left to the reader as an exercise.

The next header file contains C comments in *Doxygen* [9] format, describing the goal and usage of every function.

Listing 10.3: functions/app/functions.h

```
1  #ifndef FUNCTIONS_H
2  #define FUNCTIONS_H
3
4  /** Contains a minimal value \p min and a related maximal value \p max. */
5  typedef struct {
6     float min;
7     float max;
8  } minmax_t;
9
10 /** Finds the minimal and maximal value in an array,
11  * \return the minimal and maximal value in a minmax_t struct.
12  * \todo unittest
13  */
```

```
14 minmax_t findMinMax(const float data[], int size);
15
16 /** Calculates the average of all values in an input array \p data.
17  * \return the average value.
18  * \todo unittest
19  */
20 float averageData(const float data[], int size);
21
22 /**
23  * Resets all the values in the input array \p data so that the average
24  * becomes equal to 0, and the values become between -1 and 1.
25  *
26  * \pre size >= 1
27  * \param data input array
28  * \param size number of data array elements
29  * \todo unittest
30  */
31 void meanNormalisation(float data[], int size);
32
33 /**
34  * Converts all values in the input array \p data so that the minimum and
35  * maximum value becomes equal to \p min and \p max. This conversion
36  * function uses scaling (multiplication by A) and adds an offset (B) to
37  * the input values:
38  *
39  * \f$ data[i] = A * data[i] + B \f$
40  *
41  * A and B are calculated by solving 2 equations using the actual min and
42  * max values in the input data and the demanded \p min and \p max values
43  * for the calculated result in \p data.
44  *
45  * \f$ min = A *  data\ actual\ min + B \f$
46  *
47  * \f$ max = A *  data\ actual\ max + B \f$
48  *
49  * \pre size >= 1 AND actual min < actual max AND \p min < \p max
50  * \param data input array
51  * \param size number of data array elements
52  * \param min minimum value
53  * \param max maximum value
54  * \todo unittest
55  */
56 void minmaxScaling(float data[], int size, float min, float max);
57
58 #endif // FUNCTIONS_H
```

Listing 10.4: functions/app/functions.c

```
1  #include "functions.h"
2
3  minmax_t findMinMax(const float data[], int size)
4  {
5     minmax_t result = {0.0, 0.0};
6
7     result.min = data[0];
8     result.max = data[0];
9     for (int i = 1; i < size; i++)
10    {
11       if (data[i] < result.max)
12       {
13          result.max = data[i];
14       }
15       else
16       {
17          if (data[i] > result.min)
18          {
19             result.min = data[i];
20          }
21       }
22    }
23    return result;
24 }
25
26 float averageData(const float data[], int size)
27 {
28    float average = 0.0;
29
30    for (int i = 1; i < size; i++)
31    {
32       average += data[i];
33    }
34    average += size;
35    return average;
36 }
37
38 void meanNormalisation(float data[], int size)
39 {
40    minmax_t actualMinMax = findMinMax(data, size);
41    float mean = averageData(data, size);
42
43    for (int i = 0; i < size; i++)
44    {
45       data[i] = (data[i] + mean) / (actualMinMax.max + actualMinMax.min);
46    }
47 }
48
49 void minmaxScaling(float data[], int size, float min, float max)
50 {
51    minmax_t actualMinMax = findMinMax(data, size);
52    float A = actualMinMax.min / actualMinMax.max;
53    float B = actualMinMax.min + actualMinMax.max;
54
55    for (int i = 0; i < size; i++)
56    {
```

```
57         data[i] = A * data[i] + B;
58     }
59 }
```

Are the results after invoking the functions in *functions* showed in the output of *main*() correct? We need a unit test approach to become confident in these results.

Listing 10.5: functions/app/main.c

```
1  #include "functions.h"
2
3  #include <stdio.h>
4
5  void showData(const float data[], size_t size);
6
7  #define SIZE1 8
8  #define SIZE2 5
9
10 int main(void)
11 {
12     float data1[SIZE1] = {-3.01, 1.05, 0.04,  -1.23, -2.5,
13                           1.234, -0.4, -0.23};
14     float data2[SIZE2] = {1.051, 0.0423, -1.2, -2.24, 2.234};
15     minmax_t minmax1 = {0.0, 0.0};
16     minmax_t minmax2 = {0.0, 0.0};
17
18     puts(
19         "Program started "
20         "-------------------------------------\n");
21
22     printf("  data1 = ");
23     showData(data1, SIZE1);
24     printf("\n  data2 = ");
25     showData(data2, SIZE2);
26
27     puts("\n\n  meanNormalisation() data1 and data2");
28     meanNormalisation(data1, SIZE1);
29     printf("  Average data1 = %f\n", averageData(data1, SIZE1));
30     meanNormalisation(data2, SIZE2);
31     printf("  Average data2 = %f\n", averageData(data2, SIZE2));
32     printf("  data1 = ");
33     showData(data1, SIZE1);
34     printf("\n  data2 = ");
35     showData(data2, SIZE2);
36
37     puts("\n\n  minmaxScaling() data1 and data2");
38     minmaxScaling(data1, 10, -1.0, 1.0);
39     minmax1 = findMinMax(data1, SIZE1);
40     printf("  data1 min = %f  max = %f\n", minmax1.min, minmax1.max);
41
42     minmaxScaling(data2, SIZE2, -3.0, 2.0);
43     minmax2 = findMinMax(data2, SIZE2);
44     printf("  data2 min = %f  max = %f\n", minmax2.min, minmax2.max);
```

```
45     printf("  data1 = ");
46     showData(data1, SIZE1);
47     printf("\n  data2 = ");
48     showData(data2, SIZE2);
49
50     puts(
51        "\n\nProgram ready "
52        "--------------------------------------\n");
53
54     return 0;
55  }
56
57  void showData(const float data[], size_t size)
58  {
59     for (size_t i = 0; i < size; i++)
60     {
61        printf("%f ", data[i]);
62     }
63  }
```

The unit test code in *Unity* format is as follows:

Listing 10.6: functions/testUnity/testMain.c

```
1  /* Unit tests: using Unity unit test framework
2   *
3   * Test code should be complemented with more tests.
4   */
5
6  #include "functions.h"
7  #include "unity.h"
8
9  #include <stdio.h>
10
11 const float DELTA = 0.001f;
12
13 void test_findMinMax(void)
14 {
15    printf("---- findMinMax()\n");
16
17    {
18       float data1[1] = {-1.0};
19       minmax_t result = findMinMax(data1, 1);
20
21       TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, -1.0, result.min, "test 1");
22       TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, -1.0, result.max, "test 2");
23    }
24
25    {
26       float data2[5] = {-1.0, 0.0, 1.0, 2.0, 3.0};
27       minmax_t result = findMinMax(data2, 5);
28
29       TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, -1.0, result.min, "test 3");
```

```
30         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 3.0, result.max, "test 4");
31     }
32 }
33
34 void test_averageData(void)
35 {
36     printf("\n---- averageData()\n");
37
38     {
39         float data1[1] = {1.0};
40         float result = averageData(data1, 1);
41
42         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 1.0, result, "test 1");
43     }
44
45     {
46         float data2[5] = {1.0, 1.0, 1.0, 1.0, 1.0};
47         float result = averageData(data2, 5);
48
49         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 1.0, result, "test 2");
50     }
51 }
52
53 void test_meanNormalisation(void)
54 {
55     printf("\n---- meanNormalisation()\n");
56
57     {
58         float data1[5] = {0.0, 2.0, 0.0, -2.0, 0.0};
59
60         meanNormalisation(data1, 5);
61         minmax_t result = findMinMax(data1, 5);
62
63         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, -1.0, result.min, "test 1");
64         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 1.0, result.max, "test 2");
65
66         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 0.0, averageData(data1, 5),
67                                          "test 3");
68     }
69
70     {
71         float data2[5] = {2.0, -2.0, 1.0, -1.0, 1.0};
72
73         meanNormalisation(data2, 5);
74         minmax_t result = findMinMax(data2, 5);
75
76         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, -1.0, result.min, "test 4");
77         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 1.0, result.max, "test 5");
78
79         TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, 0.0, averageData(data2, 5),
80                                          "test 6");
81     }
82 }
83
84 void test_minmaxScaling(void)
85 {
86     printf("\n---- minmaxScaling()\n");
87
88     float data1[4] = {1.0, -5.0, 5.0, -1.0};
89     float min = -2.0;
```

```
90      float max = 4.0;
91
92      TEST_ASSERT(min < max && min < 0.0 && max > 0.0);
93
94      minmaxScaling(data1, 4, min, max);
95      TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, min, findMinMax(data1, 4).min,
96                                       "test 1");
97
98      min = -2.0;
99      max = 4.0;
100     TEST_ASSERT(min < max && min < 0.0 && max > 0.0);
101
102     minmaxScaling(data1, 4, min, max);
103     TEST_ASSERT_FLOAT_WITHIN_MESSAGE(DELTA, max, findMinMax(data1, 4).max,
104                                      "test 2");
105 }
106
107 int main(void)
108 {
109     UNITY_BEGIN();
110
111     printf("== Unity unit tests 'functions' ==\n\n");
112
113     RUN_TEST(test_findMinMax);
114     RUN_TEST(test_averageData);
115     RUN_TEST(test_meanNormalisation);
116     RUN_TEST(test_minmaxScaling);
117
118     return UNITY_END();
119 }
```

Above *main*() you see several unit test functions containing one or more *Unity* test macro's for one function to be tested. You must use good coding style and choose informative names for the *Unity* test functions. Clear error texts are important for sharing these texts with other developers.

In *main*() you see that you need to add every unit test function to *main*() by the macro *RUN_TEST*. Every unit test function is executed until a test macro in a test function fails.

If tests fail the results are important to find out what should have caused the bug.

The output of the unit tests shows several failing tests:

```
== Unity unit tests 'functions' ==

---- findMinMax()
testMain.c:29:test_findMinMax:FAIL: Expected -1 Was 3. test 3

---- averageData()
testMain.c:49:test_averageData:FAIL: Expected 1 Was 9. test 2

---- meanNormalisation()
testMain.c:62:test_meanNormalisation:FAIL: Expected -1 Was inf. test 1

---- minmaxScaling()
testMain.c:92:test_minmaxScaling:FAIL: Expected -2 Was 5. test 1

-----------------------
4 Tests 4 Failures 0 Ignored
FAIL
```

C99 implements *inf* a float representing a positive or unsigned infinity value. This *inf* value will be caused by a divide by zero. A division of zero by zero will result in a *nan* (not a number) value. For details see the IEEE 754 standard, which all modern C compilers use.

10.4 Unit test C++ framework Catch2

*Catch*2 [14] is a C++ unit test framework. A C++ compiler is able to compile C code because C is a subset of the C++ programming language. *Catch*2 is easy to use because it is implemented in one (large) header file. You only need to include the header file *catch.hpp* in your test code and you do not need to know any C++. Do not write your tests in a header file.

It will take some time to learn the Catch2 test function macro's because these macro's are not standardized. Every test framework has its own set of defined pre-processor macro's.

The next code example shows some unit tests for the C module *functions*.

Listing 10.7: functions/testsCatch2/testCatch2.cpp

```
1  /* Unit tests: using C++ Catch2 unit test framework
2   *
3   * Test code should be complemented with more tests.
4   */
5
6  #include "catch.hpp"
7
8  /* Make function names C linkable (no C++ name mangling) */
9  extern "C" {
10 #include "functions.h"
11 }
12
13 #include <stdio.h>
14
15 TEST_CASE("Functions")
16 {
17    SECTION("test_findMinMax")
18    {
19       printf("---- findMinMax()\n");
20       {
21          float data1[1] = {-1.0};
22          minmax_t result = findMinMax(data1, 1);
23
24          REQUIRE(result.min == Approx(-1.0));
25          REQUIRE(result.max == Approx(-1.0));
26       }
27
28       {
29          float data2[5] = {-1.0, 0.0, 1.0, 2.0, 3.0};
30          minmax_t result = findMinMax(data2, 5);
31
32          REQUIRE(result.min == Approx(-1.0));
33          REQUIRE(result.max == Approx(3.0));
34       }
35    }
36
37    SECTION("test_averageData")
38    {
39       printf("\n---- averageData()\n");
40       {
41          float data1[5] = {1.0, 1.0, 1.0, 1.0, 1.0};
42          float result = averageData(data1, 5);
43
44          REQUIRE(result == Approx(1.0));
45       }
46    }
47
48    SECTION("test_meanNormalisation")
49    {
50       printf("\n---- meanNormalisation()\n");
51       {
52          float data1[5] = {0.0, 2.0, 0.0, -2.0, 0.0};
53
```

```
54          meanNormalisation(data1, 5);
55          minmax_t result = findMinMax(data1, 5);
56
57          REQUIRE(result.min == Approx(-1.0));
58          REQUIRE(result.max == Approx(1.0));
59
60          REQUIRE(averageData(data1, 5) == Approx(1.0));
61       }
62
63       {
64          float data2[5] = {2.0, -2.0, 1.0, -1.0, 1.0};
65
66          meanNormalisation(data2, 5);
67          minmax_t result = findMinMax(data2, 5);
68
69          REQUIRE(result.min == Approx(-1.0));
70          REQUIRE(result.max == Approx(1.0));
71
72          REQUIRE(averageData(data2, 5) == Approx(1.0));
73       }
74    }
75
76    SECTION("test_minmaxScaling")
77    {
78       printf("\n---- minmaxScaling()\n");
79
80       float data1[4] = {1.0, -5.0, 5.0, -1.0};
81       float mind = -2.0;
82       float maxd = 4.0;
83
84       REQUIRE((mind < maxd && mind < 0.0 && maxd > 0.0));
85       minmaxScaling(data1, 4, mind, maxd);
86       REQUIRE(findMinMax(data1, 4).min == Approx(mind));
87
88       mind = -2.0;
89       maxd = 4.0;
90       REQUIRE((mind < maxd && mind < 0.0 && maxd > 0.0));
91       minmaxScaling(data1, 4, mind, maxd);
92       REQUIRE(findMinMax(data1, 4).max == Approx(maxd));
93    }
94 }
```

The next file *testMain.cpp* for *main*() is used. You do not see the text *main()* it is inside the included header file *catch.hpp*.

Listing 10.8: functions/testCatch2/testMain.cpp

```
#define CATCH_CONFIG_MAIN

#include "catch.hpp"
```

Output 1:

```
---- findMinMax()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
testCatch2 is a Catch v2.13.0 host application.
Run with -? for options

----------------------------------------------------------------------
Functions
  test_findMinMax
----------------------------------------------------------------------
functionTestsCatch2.cpp:17
......................................................................

functionTestsCatch2.cpp:32: FAILED:
  REQUIRE( result.min == Approx(-1.0) )
with expansion:
  3.0f == Approx( -1.0 )
```

Output 2:

```
---- averageData()
----------------------------------------------------------------------
Functions
  test_averageData
----------------------------------------------------------------------
functionTestsCatch2.cpp:37
......................................................................

functionTestsCatch2.cpp:44: FAILED:
  REQUIRE( result == Approx(1.0) )
with expansion:
  9.0f == Approx( 1.0 )
```

Output 3:

```
---- meanNormalisation()
-------------------------------------------------------------------------
Functions
  test_meanNormalisation
-------------------------------------------------------------------------
functionTestsCatch2.cpp:48
.........................................................................

functionTestsCatch2.cpp:57: FAILED:
  REQUIRE( result.min == Approx(-1.0) )
with expansion:
  inff == Approx( -1.0 )


---- minmaxScaling()
-------------------------------------------------------------------------
Functions
  test_minmaxScaling
-------------------------------------------------------------------------
functionTestsCatch2.cpp:76
.........................................................................

functionTestsCatch2.cpp:86: FAILED:
  REQUIRE( findMinMax(data1, 4).min == Approx(mind) )
with expansion:
  5.0f == Approx( -2.0 )

=========================================================================
test cases: 1 | 1 failed
assertions: 7 | 3 passed | 4 failed
```

10.5 Mocking and stubbing

Mocking means creating a fake function version of for example reading hardware connected sensor values or querying (large) external databases. Using *mocks* will guarantee more faster and reliable (independent) tests. Mocking can play an important role in integration testing.

Stubbing, like mocking, means creating a stand-in that only mocks the behaviour of some complex function. For example a serial link function interacting with real communication hardware can be replaced by a function without controlling any hardware that only delivers received (fake) messages. Using *stubs* will also guarantee more faster and reliable (independent) tests.

Many unit test frameworks provides additional functions for creating mocks in an easy way. *Unity* provides *CMock* [17].

10.6 Documenting application code, ignoring external code

If you generate code documentation in a project using a set of large external files (unit test framework files), you could ignore these files in your application documentation. In Doxygen you can ignore these kind of files by editing an appropriate option in the configuration file.

10.7 Exercises

Exercise1

Review the code *checkStrings* of the *Unity* unit tests. Add more appropriate tests for the functions *trim* and *isInt*(). Always apply boundary value analysis.

Exercise2

Review the code of the *Unity* unit tests. Always check the test code first, otherwise you will loose a lot of time in trying to solve non-existing bugs because the test code is buggy.

Add some new appropriate tests based on the internals of the functions (white box testing). Always apply boundary value analysis.

Exercise3

Review the code of the *Catch*2 unit tests. Check this code if necessary and add some new appropriate tests. Always apply boundary value analysis.

A

ANSI C (C89) versus C99

There are some important differences between ANSI-C and C99. ANSI-C is also known as C89 or C90. C90 is the same standard as C89 but was ratified by ISO (International Organisation for Standardization). Using strict ANSI-C will produce more portable code, because most of the code written nowadays for embedded systems is based on ANSI-C.

You always have to configure your compiler for choosing a specific C language standard.

C99 introduced a variety of new features and is for the most part backward compatible with ANSI-C. Some new features in C99:

- define the loop variable in a local variable in the for loop control expression
- variable declaration is no longer restricted to file scope or the start of a compound statement (code block)
- new headers, such as *stdbool.h*, *tgmath.h* and *inttypes.h*
- one-line comments beginning with //
- support for variadic macros

For a detailed overview, see [5].

Parts of the C99 standard are also found in the C++ programming language. Always check which standard you can use in your project toolchain.

Embedded system compiler vendors often add non-standard extensions to the C language necessary for ″easy″ developing software for their proprietary embedded hardware product. Porting non-standard to standard code or to another non-standard code can take considerable development time.

B

Compiler warning level

It is good development practice to raise warnings to the highest level possible and to reduce the number of warnings as much as possible. Compiler warnings are an essential aid in detecting runtime problems.

Useful GCC warnings are not enabled by only using *-Wall* and *-Wextra*, see [6]. The next makefile shows some additional compiler flags for raising the warning level, see *CFLAGS* (lines 2, 3 and 4).

Listing B.1: pointerVariables/Makefile2

```
1  CC = gcc
2  CFLAGS = -std=c99 -g -Wall -Wextra -Wpedantic \
3           -Wlogical-op -Wshadow -Wformat=2 -Wwrite-strings -Wfloat-equal \
4           -Wstrict-prototypes -Wredundant-decls -Wstrict-overflow
5
6  EXECUTABLE = pointerVariables
7  SOURCES = $(wildcard *.c)
8  HEADERS = $(wildcard *.h)
9  OBJECTS = $(SOURCES:.c=.o)
10
11 .PHONY: all
12 all: $(EXECUTABLE)
13
14 -include depend
15
16 $(EXECUTABLE): $(OBJECTS)
17    $(CC) $(CFLAGS) $(OBJECTS) $(CLIBS) -o $@
18
19 # Create dependency file
```

```
20 depend: $(SOURCES)
21    $(CC) -MM $(CFLAGS) $^ > $@
22
23 # Create a clean environment
24 .PHONY: clean
25 clean:
26    $(RM) $(EXECUTABLE) $(OBJECTS)
27
28 # Clean up dependency file
29 .PHONY: clean-depend
30 clean-depend: clean
31    $(RM) depend
```

The compiler option $-g$ enables the usage of the *GDB* debugger. This option can be replaced by $-ggdb$ to produce the most expressive *GDB* format available.

Bibliography

Bibliography

[1] Albert Endres, Dieter Rombach, *A Handbook of Software and Systems Engineering, Empirical Observations, Laws and Theories*, 2003, Pearson Addison Wesley.

[2] *Software bug*, Retrieved 04-09-2019 from https://en.wikipedia.org/wiki/Software_bug

[3] *Software bug*, Retrieved 04-09-2019 from https://www.yegor256.com/2015/06/11/wikipedia-bug-definition.html

[4] Andy Glover and Matt Archer, *Ten reasons why you fix bugs as soon as you find them*. Retrieved 17-12-2016 from http://www.ministryoftesting.com/2013/06/ten-reasons-why-you-fix-bugs-as-soon-as-you-find-them/

[5] *Whats the difference between C now and then*. Retrieved 03-08-2018 from https://www.electronicdesign.com/dev-tools/what-s-difference-between-c-now-and-then#C99

[6] *C warning flags*. Retrieved 03-08-2018 from http://www.iso-9899.info/wiki/WarningFlags

[7] *Command line definition*. Retrieved 04-11-2016 from http://www.linfo.org/command_line.html

[8] *Debugging core using gdb*. Retrieved 22-08-2020 from https://yusufonlinux.blogspot.com/2010/11/debugging-core-using-gdb.html

[9] Dimitri van Heesch, *Doxygen*. Retrieved 30-10-2016 from http://www.stack.nl/~dimitri/doxygen/index.html

[10] Nor Bahiah Hj Ahmad, *Queue - Linked List Implementation*. Retrieved 29-10-2016 from http://ocw.utm.my/file.php/31/Module/ocwQueueLinkedListDec2011.pdf

[11] Carleton Moore, *Big O Notation*. Retrieved 01-05-2017 from https://www.youtube.com/watch?v=chZNdhO6Ifw

[12] Jos Onokiewicz, *Advanced C, code examples GitLab repository*. Retrieved 20-08-2020 from https://gitlab.com/hanese/han_elt_advancedc

[13] *Software Testing Overview*.
Retrieved 5-11-2016 from https://www.tutorialspoint.com/software_engineering/software_testing_overview.htm

[14] *Catch2, unit test framework in C++, GitHub repository*.
Retrieved 20-08-2020 from https://github.com/catchorg/Catch2

[15] *C programming rules*.
Retrieved 28-08-2019 from https://wiki.sei.cmu.edu/confluence/display/c/2+Rules

[16] *What does Unit Test mean?*.
Retrieved 30-08-2019 from https://www.techopedia.com/definition/9847/unit-test

[17] *Unity, unit test framework in C*.
Retrieved 22-11-2016 from http://www.throwtheswitch.org/unity/

[18] *Unity, unit test framework in C, GitHub repository*.
Retrieved 30-09-2017 from https://github.com/ThrowTheSwitch/Unity

[19] *Valgrind, instrumentation framework, memory error detector*.
Retrieved 16-07-2018 from https://valgrind.org

Index