Materialpaket 02_MENT – Mental Model

C/C++, Autor: Prof. Dr.-Ing. Carsten Link

Version 1.3.1 March 3, 2019

Contents

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, Fertigkeiten zur Problemlösung):

Sie kennen die Wertebereiche und Speicheranordnung der grundelegenden C++-Datentypen und können Werte dieser Typen manipulieren und in andere Typen umwandeln.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- die Begriffe Literal, Wert, Typ einordnen
- die wichtigsten C++-Datentypen verwenden
- Umwandlungen zwischen Datentypen vornehmen oder vornehmen lassen
- mit den für die Programmierung wichtigsten Stellenwertsystemen umgehen (dual/binär, dezimal, hexadezimal)
- einfache Funktionen implementieren, welche auf nullterminierten Zeichenketten arbeiten (C-style strings)
- erläutern, wie einzelne Zeichen, Wörter und ganze Texte in Form von Zahlen in einem Programm gespeichert und verarbeitet werden können (Zahlendarstellung in Zeichensätzen und C/C++-Zahlendatentypen)
- in einem C-Programm zwischen verschiedenen Zahlendarstellungen hin- und herwandeln
- einfache Berechnungen auf Zahlen, welche in Form von Zeichenketten vorliegen, implementieren
- Bool'sche Operationen in C-Programm berechnen lassen

2 Konzepte

In den folgenden Unterabschnitten werden die Konzepte vorgestellt, welche Sie im Abschnitt *Material zum aktiven Lernen* praktisch umsetzen können.

Im folgenden werden die wichtigsten C++-Konzepte dargestellt, die der Speicherung von Daten dienen. In diesem Materialpaket geht es jedoch nur um kleinste Einheiten von Daten wie z. B. Zahlen oder Buchstaben – nicht um zusammenhängende Daten, die Strukturen oder Beziehungen untereinander aufweisen (z.B. Daten aus Steuererklärungsformularen).

Daten sind Angaben über Fakten der Realwelt oder anders gesagt: Ausprägungen von Informationen. Daten werden in Computersystemen in Form von Nullen und Einsen gespeichert. Welche Bedeutung eine Folge von Nullen und Einsen für ein C++-Programm hat, hängt von dem Datentyp ab, den man zur Interpretation des Bitmusters heranzieht.

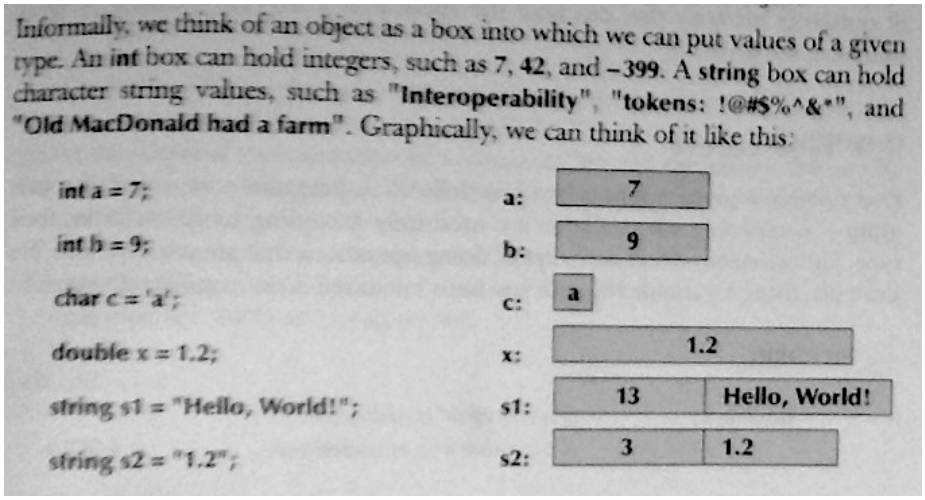
2.1 Begriffe

- Wert: ein Bitmuster im Speicher, das als Ausprägung eines Typens interpretiert wird.
- Literal: eine Zeichenkette im Quelltext, die einen konkreten Wert darstellt. Beispielsweise stellt `0.0` die Fließkommazahl mit dem Wert Null dar und das Literal `false` stellt den Wahrheitswert falsch dar.

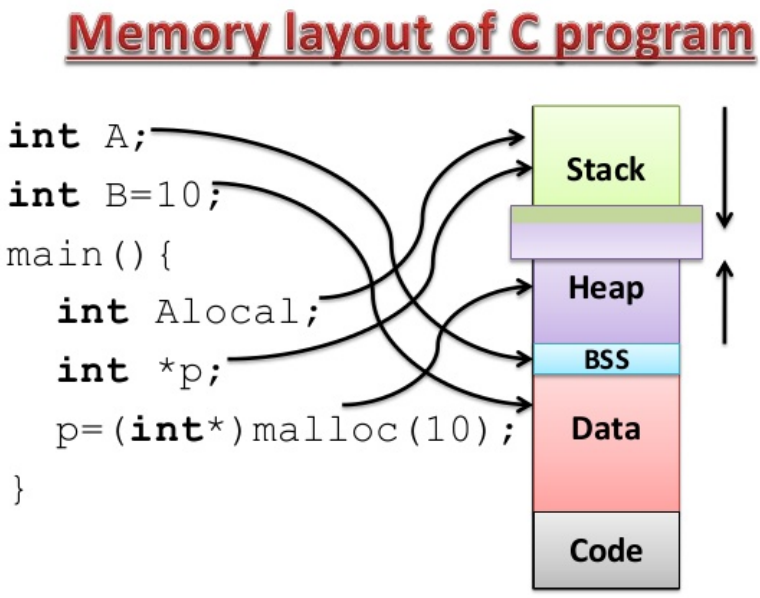
[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen

- Datentyp (Typ, engl. type or data type): eine Menge von möglichen gleichartigen Werten (und Operatoren darauf). Beispielsweise umfasst der Datentyp `bool` die Werte `true` und `false`. Mit Typen geht einher, wie Bitmuster im Speicher als Wert zu interpretieren sind
- Eingebaute Datentypen (built-in types): Die wichtigsten in C++ eingebauten Datentypen (auch primitive Datentypen genannt) sind (siehe [PPP] A.8):
  - `bool`: für Wahrheitswerte
  - `char`: für Zeichen oder Bytes
  - `int, long`: für (Integers, signed / unsigned)
  - `double`: für Fließkommazahlen
- Variablen: lokal, global, Sichtbarkeit, Gültigkeit, Initialisierung, Speicherort (stack, heap, rodata, bss), Namensgebung
- Aggregatstyp array: Zusammenfassung vieler gleichartiger Datenelemente (Zugriff per Index)
- Aggregatstyp struct: Zusammenfassung vieler verschiedenartiger Datenelemente (Zugriff per Name)

Definitionen der Begriffe type, value, variable, declaration, definition etc. siehe [PPP] §3.8.



Werte und deren Abbild im Speicher (vereinfacht, aus [PPP] S. 77). Beachten Sie den Unterschied zwischen `1.2` und `"1.2"`.



Die verschiedenen Speicherbereiche zur Laufzeit mit den jeweils aufgenommenen C-Konstrukten. Quelle[2].

Zur Laufzeit werden die C-Konstrukte wie folgt im Speicher abgelegt:

- *Text* oder *Code*: ausführbarer Maschinencode
- *Data*: globale initialisierte Variablen
- *BSS*: globale nicht-initialisierte Variablen (wird vom Betriebssystem mit Nullwerten vorbelegt)
- *Heap*: zur Laufzeit angeforderte Datenbereiche (`malloc()`, `new`-Operator); wächst tendenziell nach oben; wird ggf. löchrig (Freigabe per `free()` bzw. `delete`-Operator)
- *Stack*: Call Stack zur Aufnahme der Aktivierungsrecords; wächst nach unten

Die Bereiche *Text, Data, BSS* sind zur Laufzeit nicht mehr in der Größe änderbar. Die Bereiche *Stack* und *Heap* werden vom Betriebssystem größer als benötigt angelegt, so dass hier zur Laufzeit dynamisch die Größe des tatsächlich verwendeten Speichers angepasst werden kann (Details folgen in späteren Kapiteln).

Die Besonderheiten des Speicherlayouts in eingebetteten Systemen (Microkontroller) werden in dem Artikel [FLSH] beleuchtet.

[2] https://www.slideshare.net/siddguruk/lec06-44691468

2.2 Zahlendarstellung und Wertebereiche in C/C++

In diesem Abschnitt werden die wichtigsten primitiven Datentypen behandelt.

Relationship with C library macro constants

Specialization	Members						
	min()	lowest() (C++11)	max()	epsilon()	digits	digits10	min_e
numeric_limits< bool >							
numeric_limits< char >	CHAR_MIN	CHAR_MIN	CHAR_MAX				
numeric_limits< signed char >	SCHAR_MIN	SCHAR_MIN	SCHAR_MAX				
numeric_limits< unsigned char >	0	0	UCHAR_MAX				
numeric_limits< wchar_t >	WCHAR_MIN	WCHAR_MIN	WCHAR_MAX				
numeric_limits< char16_t >	0	0	UINT_LEAST16_MAX				
numeric_limits< char32_t >	0	0	UINT_LEAST32_MAX				
numeric_limits< short >	SHRT_MIN	SHRT_MIN	SHRT_MAX				
numeric_limits< signed short >							
numeric_limits< unsigned short >	0	0	USHRT_MAX				
numeric_limits< int >	INT_MIN	INT_MIN	INT_MAX				
numeric_limits< signed int >							
numeric_limits< unsigned int >	0	0	UINT_MAX				
numeric_limits< long >	LONG_MIN	LONG_MIN	LONG_MAX				
numeric_limits< signed long >							
numeric_limits< unsigned long >	0	0	ULONG_MAX				
numeric_limits< long long >	LLONG_MIN	LLONG_MIN	LLONG_MAX				
numeric_limits< signed long long >							
numeric_limits< unsigned long long >	0	0	ULLONG_MAX				
numeric_limits< float >	FLT_MIN	-FLT_MAX	FLT_MAX	FLT_EPSILON	FLT_MANT_DIG	FLT_DIG	FLT_M
numeric_limits< double >	DBL_MIN	-DBL_MAX	DBL_MAX	DBL_EPSILON	DBL_MANT_DIG	DBL_DIG	DBL_M
numeric_limits< long double >	LDBL_MIN	-LDBL_MAX	LDBL_MAX	LDBL_EPSILON	LDBL_MANT_DIG	LDBL_DIG	LDBL_

Wertebereiche von Ganzzahl- und Fließkommazahl-Datentypen (Templates und Macros aus dem `limits`-Header und `limits.h` Quelle: http://en.cppreference.com/w/cpp/types/numeric_limits)

Das folgende Programm verdeutlicht, wie die Grenzen der Wertebereiche ermittelt werden können (der `sizeof`-Operator liefert die Größe seines Operanden in Bytes zurück):

```
1  #include "../../../helpers/println.hpp"
2  #include <limits>
3
4  int main()
5  {
6    println("type\t sizeof\t\t lowest\t\t\t\t highest\n");
7    println("float \t ",sizeof(float), "\t\t\t",
8            std::numeric_limits<float>::lowest(), "\t\t",
9            std::numeric_limits<float>::max());
10   println("double\t ",sizeof(double), "\t\t\t",
11           std::numeric_limits<double>::lowest(), "\t\t",
12           std::numeric_limits<double>::max());
```

```
13    println("int   \t ",sizeof(int), "\t\t\t",
14            std::numeric_limits<int>::lowest(), "\t\t\t",
15            std::numeric_limits<int>::max());
16    println("long  \t ",sizeof(long), " \t\t",
17            std::numeric_limits<long>::lowest(), '\t',
18            std::numeric_limits<long>::max());
19
20    }
```

Die Ausgabe könnte wie folgt aussehen:

```
type      sizeof      lowest               highest

float     4           -3.402823e+38        3.402823e+38
double    8           -1.797693e+308       1.797693e+308
int       4           -2147483648          2147483647
long      8       -9223372036854775808     9223372036854775807
```

Da diese Werte von der Hardwarearchitektur, dem Betriebsystem und dem Compiler abhängen können, ist es gelegentlich sinnvoll, integrale Datentypen mit fest vorgegebener Größe zu verwenden. So finden sich in `<stdint.h>` neben `int16_t` für 16-Bit breite vorzeichenbehaftete Ganzzahlen und `uint64_t` für 64-Bit breite positive Ganzzahlen noch viele weitere Typdeklarationen und Makro-Konstanten wie z. B. `INT32_MIN`.

Hinweis: das obige Programm verwendet aus der Header-Datei `helpers/println.hpp` die spezielle Funktion `println()`, welche einfacher für textuelle Ausgaben zu verwenden ist, als die üblichen C++-Standardausgabeverfahren (Streams mit `std::cout`). Beispiele hierzu finden sich in der Datei `printlnDemo.cpp`. Darin ist zu sehen, wie die Funktion mit mehreren verschiedenen Parametern aufgerufen wird.

2.3 Bool'sche Operationen zur Bitmanipulation

In C++ ist es möglich, logische Operationen Bit-weise anwenden zu lassen. So lässt sich zum Beispiel prüfen, ob in einem `int`-Wert bestimmte bits gesetzt sind. Der folgende Code zeigt die Verwendung solcher binärer boolscher Operatoren:

```
1  int a = 0xfe;         // a=254; a=0b1111'1110;
2  int b = 0x07;         // b=7;   b=0b0000'0111;
3  int a_OR_b  = a | b;  // 0b1111'1111
4  int a_AND_b  = a & b;  // 0b0000'0110
5  int a_XOR_b = a ^ b;  // 0b1111'1001
```

Oben enthält die Variable `a_OR_b` das Ergebnis der Bit-für-Bit-Oder-Verknüpfung der Variablen `a` und `b` (siehe Kommentare); die Variable `a_AND_` enthält das Ergebnis der Bit-für-Bit-Und-Verknüpfung der Variablen `a` und `b`; die Variable

`a_XOR_b` enthält das Ergebnis der Bit-für-Bit-exklusiv-Oder-Verknüpfung der Variablen `a` und `b`.

Insbesondere bei der der Implementierung von Netzwerkprotokollen, kryptographischen Algorithmen, Dateiformaten oder der hardwarenahen Programmierung (Mikrocontroller) ist es oftmals notwendig, gezielt einzelne Bits zu manipulieren (setzen, löschen, ...) – ohne jedoch die anderen Bits eines Bytes zu verändern. Beispielsweise um die Übertragungsmodi einer seriellen Schnittstelle[3] einzustellen, müssen einzelne Bits entsprechend gesetzt werden (Bits 3, 4 und 5 im Line Control Register des UART 16550[4]).

Aufgrund der Wahrheitstabelle der logischen Bitoperationen ergeben sich folgende Rechentricks:

- Setzen eines oder mehrerer Bits: *oder* mit Muster der zu setzenden Bits
- Löschen eines oder mehrerer Bits: *und* mit invertierten Muster der zu löschenden Bits
- Umdrehen eines oder mehrerer Bits: *exklusiv oder* mit Muster der zu drehenden Bits
- Multiplizieren mit zwei: einmal nach Links schieben
- Dividieren durch zwei: einmal nach Rechts schieben
- Divisionsrest: der Modulo-Operator liefert den Rest einer ganzzahligen Division (`13 % 4` ergibt `1`). Hierbei nimmt das Ergebnis von `m % n` Werte im Bereich `0..n-1` (einschließlich) an

```
int controlRegister   = 128;       // bit number 7 is set
controlRegister      |= 64 + 32;   // set bits #6, #5
controlRegister      ^= 16;        // reverse bit #4
controlRegister      &= 64         // clear all bits except #6
controlRegister      >>= 1         // shift right, i.e. divide by two
```

2.4 Darstellung von Texten

Computer können im Wesentlichen nur Zahlen verarbeiten (Nullen und Einsen). Um üblicherweise anders dargestellte Daten verarbeiten zu können, müssen diese Daten in Form von Bitmustern oder Zahlen dargestellt werden. Zur Darstellung von Buchstaben, Ziffern, Satz- und Sonderzeichen werden Tabellen verwendet (d. h. die Zeichen sind durchnummeriert).

Aus einzelnen Zeichen können Zeichenketten zusammengestellt werden, um Wörter, Sätze und ganze Texte in Form von Zahlen im Computer zu speichern und zu verarbeiten:

- Zeichen: `a`, `b`, `c`, ..., `A`, ..., `Z`, `0`, ..., `9`

[3] https://en.wikipedia.org/wiki/16550_UART

[4] https://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming#Line_Control_Register

- Zeichenketten: `Franz jagt im komplett verwahrlosten Taxi quer durch Bayern`, `12`, `zwölf`

Die traditionelle Tabelle, die druckbare Zeichen mit Zahlen verknüpft, ist die ASCII-Tabelle. Überfliegen Sie den Wikipedia-Artikel ASCII[5], um ein grobes Verständnis der Zuordnung von druckbaren Zeichen (Buchstaben, Ziffern, Satz- und Sonderzeichen) und Zahlenwerten zu bekommen. Besonders hervorzuheben ist die Zuordnung der druckbaren Ziffer `0` und dem dazugehörigen Zahlenwert `48`.

Wichtige Zeichen der ASCII-Tabelle (siehe auch[6]):

Dezimalwert	Zeichen	isalpha()	isdigit()	isalnum()	iscntl()
9	TAB	false	false	false	true
10	LF	false	false	false	true
13	CR	false	false	false	true
32	SPACE	false	false	false	false
48	0	false	true	true	false
49	1	false	true	true	false
57	9	false	true	true	false
65	A	true	false	true	false
66	B	true	false	true	false
67	C	true	false	true	false
97	a	true	false	true	false
98	b	true	false	true	false
99	c	true	false	true	false

Dabei steht SPACE für das Leerzeichen, CR für Carriage Return und LF für Line Feed. Die letzten beiden waren früher bei mechanischen Druckern wichtig, da sie den Druckkopf bzw. den Papiervorschub steuerten. Heutzutage finden sie noch Verwendung als Zeilenumbruch, wobei hier verschiedene Konventionen üblich sind, die von dem verwendeten Betriebssystem oder Protokoll abhängen (siehe[7]; Dos/Windows, HTTP: CRLF, Unix: LF).

Obwohl der ASCII-Zeichensatz kaum noch Verwendung findet, ist er wichtig, da er die Grundlage für andere Zeichensätze bildet. In wesentlichen Teilen (Ziffern, Buchstaben und White Spaces) stimmen heutige Zeichensätze (ISO-8859-15, Unicode, ...) mit ASCII überein.

[5] https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

[6] http://www.asciitable.com

[7] https://de.wikipedia.org/wiki/Zeilenumbruch

2.5 Nullterminierte Strings

Da die althergebrachten C-artigen Zeichenketten auch heute noch weit verbreitet sind und ihre Daseinsberechtigung haben, können Sie hier nicht unerwähnt bleiben.

Bei Zeichenketten ist vom Programmierer oftmals nicht vorherzusagen, wie lang sie sein werden und wieviele davon benötigt werden (beides trifft auf viele Datenstrukturen in unterschiedlicher Stärke auf).

Die Grundidee für nullterminierte Zeichenketten ist einfach, hat jedoch nicht ganz so offensichtliche Konsequenzen. Statt für jede zu speichernde Zeichenkette einen `int`-Wert mit dessen Länge zu speichern, wird das `char`-Symbol mit dem `int`-Wert `0` als Endekennung interpretiert. Der String `Hello, world!` wird also als `'H', 'e', 'l', 'l', ... 'd', '!', 0` gespeichert (beachten Sie, dass bei der `0` die einfachen Anführungsstriche fehlen, da es sich um einen `int`-Literal und nicht um einen `char`-Literal handelt).

Wichtige Konsequenzen davon sind:

- es muss ein `char` mehr als nötig allokiert werden
- die Länge muss durch Iteration ermittelt werden; sie kann nicht ausgelesen werden
- Änderungen an Zeichenketten, die deren Länge ändern, ziehen in der Regel (Re-) Allokation und Duplikation nach sich (umkopieren in neuen Speicherbereich)
- Operationen auf Zeichenketten sind üblicherweise mit Zeigerarithmetik und dynamischem Speicher zu implementieren (sehr fehleranfällig)

2.6 Pascal-Strings

Bei der Programmiersprache Pascal ist es traditionell üblich, Zeichenketten nicht mit einer Endekennung (Null-terminiert) zu speichern. Hier wird zusätzlich zu den Zeichen die Länge der Zeichenkette gespeichert. Der String `Hello, world!` wird also als ‘13, ‘H’, ‘e’, ‘l’, ‘l’, . . . ‘d’, ‘!’ . . . gespeichert.

Wichtige Konsequenzen davon sind:

- Die maximale Länge von Zeichenketten ist beschränkt auf den maximal darstellbaren Wert im Längenfeld. Wird nur eine Byte vorgesehen, so dürfen Zeichenketten nicht länger als 255 Zeichen sein
- Zu jeder Zeichenkette wird das Längenfeld gespeichert, was ein bis acht zusätzliche Bytes ausmachen kann

2.7 Zahlendarstellung in ASCII

Daten, die von einem Programm verarbeitet werden sollen, stammen in der Regel aus externen Quellen (Tastatur, Datei, Netzwerk) und liegen in Textform vor. Das heißt: auch Zahlen sind als ASCII-Zeichen kodiert. Um diese Zahlen verarbeiten zu können, müssen sie erst in `int` oder `double`-Werte umgewandelt werden.

Die folgende Funktion wandelt die in ASCII-Kodierung gegebene Zahl `string_value` in einen `unsigned int` um:

```
unsigned int asciToInt(string string_value){
    unsigned int result = 0;
    for(size_t i=0; i<string_value.length();i++){
        result *= 10;
        result += string_value[i] - 48;
    }
    return result;
}
```

Der Wert `48` ist der ASCII-Wert des Zeichens `0`. Wird 48 von einer Ziffer abgezogen, so erhält man den korrespondierenden Ganzzahlwert. Da das dezimale Zahlensystem ein Stellenwertsystem ist, muss noch jede Ziffer mit der Wertigkeit der jeweiligen Position multipliziert werden (`*= 10` ergibt die Faktoren 10, 100, 1000, ...), bevor alle Ziffernwerte aufaddiert werden (`result +=`).

2.8 Exkurs: Bitmuster im Speicher

Der Compiler legt Variablen im Speicher an. Für Programmierer ist es wichtig, eine Vorstellung davon zu haben, wie Werte im Speicher abgelegt werden. Um dies zu illustrieren, wird nun der Hilfsdatentyp `TypedMemory` vorgestellt:

```
struct TypedMemory {
  static const int RAWMEMORYSIZE=128;//1024;
  uint8_t rawMemory[RAWMEMORYSIZE];

  void putChar(int position, unsigned char c); // position in bytes starting at 0
  unsigned char getChar(int position);
  //  void putShort(int position);
  //  char getShort(int position);
  void putUInt(int position, unsigned int i);
  unsigned int getUInt(int position);
  void putDouble(int position, double d);
  double getDouble(int position);
  void putAnything(int position, void* src, int size);
  void getAnything(int position, void* dest, int size);
```

```
15    std::string hexDump();
16  };
```

In einer Variablen des Typs `TypedMemory` können mittels der `put...()` und `get...()`-Funktionen Werte abgelegt und wieder ausgelesen werden. Die Speicherung findet intern in dem Byte-Array `rawMemory[]` statt. In `main()` wird nun eine solche Variable verwendet:

```
1   int main(int argc, const char * argv[]) {
2     TypedMemory mem;
3
4     mem.putUInt(0, 0x04030201);
5     mem.putChar(4,'C');
6     mem.putChar(5,'+');
7     mem.putChar(6,'+');
8     mem.putChar(7,'!');
9     mem.putDouble(0x30, 16.0 /*355.0/113.0*/);
10
11    std::cout << std::setbase(16);
12    std::cout << "0x" << mem.getUInt(0) << std::endl;
13    std::cout << mem.getChar(4)
14              << mem.getChar(5)
15              << mem.getChar(6)
16              << mem.getChar(7) << std::endl;
17    std::cout << "0x" << mem.getUInt(4) << std::endl;
18    std::cout << mem.getDouble(0x30) << " (" << sizeof(double) << " bytes)" << std::endl;
19    std::cout << mem.hexDump() << std::endl;
20
21    return 0;
```

Die Ausgabe ist diese:

```
0x4030201
C++!
0x432b2b21
16 (8 bytes)
0000:  04 03 02 01 43 2b 2b 21 20 f7 bf 5f ff 7f 00 00 ....C++! ..._....
0010:  01 02 03 04 00 00 00 00 00 00 00 00 00 00 30 40 ..............0@
0020:  40 f7 bf 5f ff 7f 00 00 01 00 00 00 00 00 00 00 @..._..........
0030:  00 00 00 00 00 00 30 40 00 00 00 00 01 00 00 00 ......0@........
0040:  00 00 00 00 00 00 00 00 68 f7 bf 5f ff 7f 00 00 ........h..._....
0050:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0060:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070:  00 00 00 00 00 00 00 00 48 f7 bf 5f ff 7f 00 00 ........H..._....
```

Es ist zu sehen, dass der `unsigned int`-Wert `0x04030201` ab der Speicherstelle `0000` abgelegt ist. Da die Variable `mem` eine lokale Variable ist, enthält sie auch zufällige Werte – daher müssen lokale Variablen immer initialisiert werden.

Die Funktion `TypedMemory::putUInt(int position, unsigned int i)` zeigt, wie ein `unsigned int i` konkret an der Stelle `position` in `rawMemory[]` abgelegt wird:

```
1  void TypedMemory::putUInt(int position, unsigned int i){
2    int numBytes = sizeof(i);
3    unsigned int byteMask = 0xff << ((numBytes-1)*8);
4    for(int k=0; k<numBytes;k++){
5      rawMemory[position+k] = (i & byteMask) >> ((numBytes-1-k)*8);
6      byteMask >>= 8;
7    }
8    /* equvalent code for sizeof(unsigned int) == 4, big endian:
9     uint8_t byte_a = (i &       0xff);
10    uint8_t byte_b = (i &     0xff00) >> 8;
11    uint8_t byte_c = (i &   0xff0000) >> 16;
12    uint8_t byte_d = (i & 0xff000000) >> 24;
13
14    rawMemory[position+0] = byte_d;
15    rawMemory[position+1] = byte_c;
16    rawMemory[position+2] = byte_b;
17    rawMemory[position+3] = byte_a;
18    */
19 }
```

Im auskommentierten Code wird ersichtlich, wie vier mal (`byte_a` bis `byte_d`) jeweils genau ein Byte aus `i` herausgeschnitten (mit `&`) und zurechtgerückt (mit `>>`) wird. Die vier einzelnen Bytes werden nun nacheinander ab `rawMemory[position]` abgelegt.

3 Material zum aktiven Lernen

Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C++-Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Nehmen unten angegebenen Code (`main_02_MENT.cpp` im Moodle-Kurs) und ergänzen sie folgendes:

- Die Funktion `hexDigitToInt()` soll für die `char`-Werte `'0'` bis `'9'` die

`int`-Werte `0` bis `9` liefern, sowie für `'a'` bis `'f'` 10 bis 15 (also eine ASCII-Hexadezimalziffer in einen korrespondierenden Zahlenwert umwandeln)
- Fügen Sie den Rumpf zur Funktion `hexStringToInt(PascalString)` hinzu, so dass als Zeichenketten gegebene Hexadezimalzahlen in einen korrespondierenden Zahlenwert umgewandelt werden

Hinweis: Der Zugriff auf die einzelnen Elemente einer `struct` erfolgt über deren Namen. Beispielsweise könnte die Initialisierung von `s2` auch wie folgt geschehen:

```
PascalString s2;
s2.length = 4;
s2.characters[0] = 'f';
s2.characters[1] = 'f';
s2.characters[2] = 'f';
s2.characters[3] = 'f';
```

Hinweis: Da es sich um C++-Code handelt, sollten Sie mit `clang++ -std=c++14` übersetzen.

```
// file: main_02_MENT.cpp
// THIS IS C++, use clang++

#include "../helpers/println.hpp"
#include <iostream>

struct PascalString{
    int length;             // number of chars used
    char characters[256];  // chars of some character string
};

int hexDigitToInt(char hexDigit){
int value = -1;
// TBD
return value;
}

int hexStringToInt(PascalString binaryDigits){
int returnValue = -1;

return returnValue;
}

int main(int argc, char** argv, char** envp){
    PascalString s = {3, '1', '0', '0'};
    PascalString s2 = {4, 'f', 'f', 'f', 'f'};
  println(hexStringToInt(s));
  println(hexStringToInt(s2));
  return 0;
```

```
30 }
```

Obiger Code verwendet die Komfortfunktion `println()`, welche einfacher zu verwenden ist als die C++-Streams (`cout << 23 << endl;`). Es wird empfohlen, diese Funktion zu verwenden.

3.2 Aufgabe: Modifikationen

Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält.

Modifikationen:

1. erstellen Sie eine Funktion `printPascalString(PascalString s)`, welche die in `s` enthaltenen Zeichen nacheinander einzeln auf der Konsole ausgibt (`println()` oder `std::cout`)
2. erstellen Sie eine Funktion `intToDual(int n)`, welche das Argument `n` in einen korrespondierenden binären `PascalString` umwandelt (z. B. `5` in `00000101`)
3. erstellen Sie eine Funktion `intToHex(int n)`, welche das Argument `n` in einen korrespondierenden hexadezimalen `PascalString` umwandelt
4. erstellen Sie eine Funktion `PascalString bitwiseDualAnd(PascalString a, PascalString b)`, welche die beiden Dualstrings `a` und `b` Bit für Zeichen für Zeichen logisch und-Verknüpft
5. erstellen Sie eine Funktion `PascalString bitwiseHexAnd(PascalString a, PascalString b)`, welche die beiden Hexadezimalstrings `a` und `b` logisch und-Verknüpft und das Resultat zurückgibt
6. Fügen Sie die globale Variable `char cStringArea[1024];` hinzu. Erstellen Sie eine Funktion `to_c_string(PascalString s)` , die eine Freie Stelle in diesem Array findet, dort einen nullterminierten String ablegt und dessen Index (Postion) zurückgibt. Der Einfachheit halber können Sie Annehmen, dass Bereiche mit den Werten `255` oder `0` frei sind

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels “Konzepte”, der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Vergleichen Sie die beiden Arten, Zeichenketten zu speichern, wie sie von Pascal und C verwendet werden. Welche Vor- und Nachteile existieren jeweils?

2. Welches Bitmuster ergibt sich am Ende des folgenden Codes?

```
int controlRegister   = 128;
controlRegister      |= 64 + 32;
controlRegister      ^= 16;
controlRegister      &= 128+64;
controlRegister      <<= 1;
```

4. Welchen Typ haben die unten angegebenen Ausdrücke?

```
1
1.0
"1.0"
1 + 1.0
'1'
```

4 Nützliche Links

- Stellenwertsystem von Martin Freidank, Sönke Greve & Felix Graf[8]
- A Tutorial on Data Representation Integers, Floating-point Numbers, and Characters https://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html
- Aggregate type in FOLDOC[9]
- JOEL ON SOFTWARE, The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/
- A tutorial on character code issues: https://www.cs.tut.fi/~jkorpela/chars.html
- Unicode In Python, Completely Demystified: http://farmdev.com/talks/unicode/

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition
- [FLSH] StratifyLabs, RAM/Flash Usage in Embedded C Programs[10]

[8] http://userpages.uni-koblenz.de/~proedler/bac/stellen.php
[9] http://www.dictionary.com/browse/aggregate-type
[10] https://stratifylabs.co/embedded%20design%20tips/2013/10/18/Tips-RAM-Flash-Usage-in-Embedded-C-Programs/