C/C++ Materialpaket (Level C)
03b_DATA – Data Representation

Prof. Dr. Carsten Link

Zusammenfassung

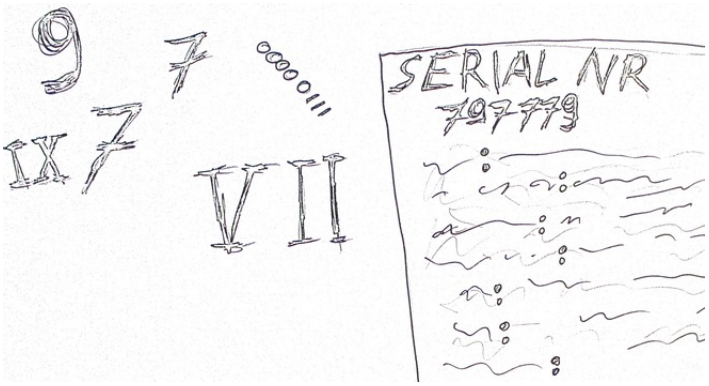


Abbildung 1: Bits, Bytes, Daten und Typen

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und 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]:

- den Begriff Typ erläutern
- die wichtigsten C++-Datentypen verwenden
- Umwandlungen zwischen Datentypen selbst umsetzen
- 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 Bitoperationen in C++-Programm einsetzen

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

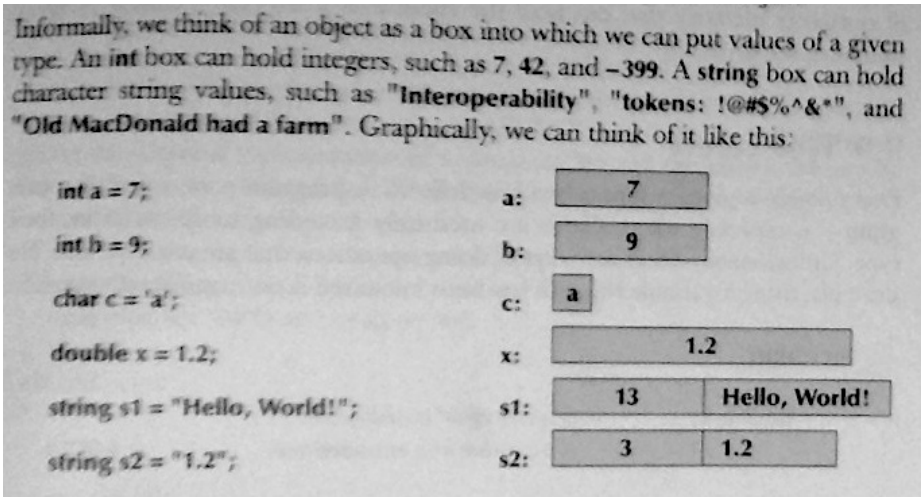
[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

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
- **Datentyp:** 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:** die wichtigsten in C++ eingebauten Datentypen (auch primitive Datentypen oder built-in types genannt) sind (siehe [PPP] A.8):
  - `bool`: für Wahrheitswerte
  - `char`: für Zeichen oder Bytes
  - `int, long`: für (Integers, signed / unsigned)
  - `double`: für Fließkommazahlen
- **Variablen:** auf Speicherstellen für Werte kann jeweils über einen Namen zugegriffen werden. Im Verlauf der Materialpakete werden diese Begriffe erläutert: lokal, global, Sichtbarkeit, Gültigkeit, Initialisierung, Speicherort, Namensgebung
- **Aggregatstyp:** Zusammenfassung vieler gleichartiger oder verschiedenartiger Datenelemente

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"`.

2.2 Aggregatstypen

C++ bietet die Möglichkeit, mehrere Elemente zusammenzufassen:

Aggregatstyp array: Zusammenfassung vieler gleichartiger Datenelemente, wobei der Zugriff per Index geschieht. Im nachfolgenden Beispiel wird ein Array von 1024 `int`-Werten definiert und verwendet:

```
#define BUFFER_SIZE 1024          // C-style constant
int buffer[BUFFER_SIZE];          // C-style array

int main(){
  buffer[0] = 0;
  buffer[BUFFER_SIZE-1] = 0;
  return 0;
}
```

In obigem Code werden das erste und das letzte Element mit dem Wert `0` beschrieben.

Aggregatstyp Struct: Zusammenfassung vieler verschiedenartiger Datenelemente, wobei der Zugriff per Name geschieht. Der Aggregatstyp Struct wird auch Record, Verbundstyp oder Datensatz genannt. Ein Beispiel:

```
struct Date {
  unsigned char day;
  unsigned char month;
  int year;
};
```

```
6
7  int main(){
8    Date EinsteinsBirthday = {14, 03, 1879};
9    int EinsteinsBirthYear = EinsteinsBirthday.year;
10   return 0;
11 }
```

In obigem Code wird eine Variable namens `EinsteinsBirthday` initialisert. Der Zugriff auf einzelne Elemente erfolgt mit dem `.`-Operator.

Kombinierte Aggregatstypen: Hier zeigt sich die Orthogonalität, die an vielen (aber nicht allen) Stellen der Sprache C++ zu finden ist: die Aggregatstypen Struct und Array lassen sich kombinieren:

```
1  struct DateWithMon {
2    unsigned char day;
3    unsigned char month;
4    int year;
5    char mon[4];
6  };
7
8  void combined(){
9    DateWithMon EinsteinsBirthday = {14, 03, 1879, {'M', 'a', 'r', 0}};
10   DateWithMon EulersBirthday = {15, 04, 1707, {'A', 'p', 'r', 0}};
11
12   DateWithMon importantBirthdays[2]; // array of structs
13   importantBirthdays[0] = EinsteinsBirthday;
14   importantBirthdays[1] = EulersBirthday;
15
16   println("[0].year = " , importantBirthdays[0].year, " mon = ", importantBirthdays[0].mon);
17   println("[1].year = " , importantBirthdays[1].year, " mon = ", importantBirthdays[1].mon);
18 }
```

Der obige Code erzeugt folgende Ausgabe:

```
EinsteinsBirthYear = +1879
[0].year = +1879 mon = Mar
[1].year = +1707 mon = Apr
```

2.3 Operationen zur Bitmanipulation

In C++ ist es möglich, logische Operationen Bit-weise anwenden zu lassen. Hier ein Überblick über alle 16 möglichen binären Bit-Operatoren:

	bit combinations				name	C/C++
a	0	1	0	1		
b	0	0	1	1		
result mask	0	0	0	0		0
result mask	0	0	0	1	AND	a & b
result mask	0	0	1	0		
result mask	0	0	1	1		b
result mask	0	1	0	0		
result mask	0	1	0	1		a
result mask	0	1	1	0	XOR	a ^ b
result mask	0	1	1	1	OR	a | b
result mask	1	0	0	0	NOR	
result mask	1	0	0	1	EQUAL	
result mask	1	0	1	0		
result mask	1	0	1	1	IMPL	
result mask	1	1	0	0		
result mask	1	1	0	1		
result mask	1	1	1	0	NAND	
result mask	1	1	1	1		1

Von den oben aufgeführten möglichen binären Bitoperatoren sind nicht alle sinnvoll; daher verfügen nur wenige über einen Namen und/oder eine Entsprechung in C/C++. Umgangssprachlich werden folgende Namen verwendet: `AND` = und, `OR` = und/oder, `XOR` = entweder oder, `NOR` = weder noch.

Neben den oben angegebenen Operationen gibt es noch Operatoren, mit denen Bits in einem integralen Wert nach links oder rechts verschoben werden können: `char eight = 1 << 3; int zero= 1 >> 1`.

Mit binären Bitoperatoren 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_b` 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. Um Beispielsweise die Übertragungsmodi einer seriellen Schnittstelle[2] einzustellen, müssen einzelne Bits entsprechend gesetzt werden (Bits 3, 4 und 5 im Line Control Register des UART 16550[3]).

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

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

2.4 Darstellung von Farben

Eine weit verbreitete Art, Farben in Porgrammen und Hardware zu kodieren, ist die RGB-Darstellung. Hierbei werden für die additive Farbmischung Rot-, Grün- und Blauwerte gespeichert.

Der nachfolgende Code zeigt, wie RGB-Werte in effizient in einem 32-Bit Integer gespeichert werden können:

```
1 #include "../helpers/println.hpp"
2 #include <cstdint> // provides fixed width integer types e.g. uint8_t
3
4 uint32_t fromRGB(uint8_t red, uint8_t green, uint8_t blue){
```

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

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

```
  return (red << 16) | (green<<8) | (blue);
}

uint8_t red(uint32_t rgb){
  return (rgb & 0xFF0000) >> 16;
}

uint8_t green(uint32_t rgb){
  return (rgb & 0xFF00) >> 8;
}

uint8_t blue(uint32_t rgb){
  return (rgb & 0xFF);
}

int main(/*int argc, char** argv, char** envp*/){
  uint32_t pureRed = fromRGB(255,0,0);
  println("pure red is ", pureRed);
  println("pure blue is ", fromRGB(0,0,255));
  println("red part of pure red is ", red(pureRed) );
  return 0;
}
```

2.5 Darstellung von Texten

Computer können im Wesentlichen nur Zahlen verarbeiten (Bytes bestehend aus 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`
- 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[4], 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`.

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

Wichtige Zeichen der ASCII-Tabelle (siehe auch[5]):

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

In obiger Tabelle 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[6]; 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.

2.6 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

[5] http://www.asciitable.com
[6] https://de.wikipedia.org/wiki/Zeilenumbruch

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)

2.7 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:

- Zu jeder Zeichenkette wird das Längenfeld gespeichert, was ein bis acht zusätzliche Bytes ausmachen kann
- Die maximale Länge von Zeichenketten ist beschränkt auf den maximal darstellbaren Wert im Längenfeld. Wird nur ein Byte vorgesehen, so dürfen Zeichenketten nicht länger als 255 Zeichen sein
- auch kurze Zeichenketten belegen im Speicher die Maximallänge
- Operationen sind meist sehr effizient umzusetzen
- auch der Zahlenwert `0` ist als Zeichen zulässig

2.8 C/C++-Strings

Die Programmiersprache C arbeitet ausschließlich mit nullterminierten Zeichenketten. Ein

```
1 char* s="abc";
```

deklariert einen nullterminierten String mit dem Namen `s`, welcher als Wert die Adresse (Speicherstelle) des ersten `char`s `'a'` hat. Mit eckigen Klammern kann auf einzelne Zeichen zugegriffen werden. Der Compiler wertet den Literal aus und legt `abc` in den Speicher. Speicher von Strings, welche erst zur Laufzeit entstehen, muss vom Programm (Programmierer) verwaltet werden.

Da nullterminierte Strings unhandlich und fehleranfällig sind, unterstützt C++ Zeichenketten mit dem Typ `std::string`. Dieser kann sich vorgestellt werden, wie eine Mischform aus nullterminierten Strings und PascalStrings. Werte und

Variablen von `std::string` sind also genauso leicht zu handhaben wie beispielsweise `int`-Werte und Variablen und haben nicht die Nachteile von PascalStrings. Der C++-Compiler unterstützt nach wie vor nullterminierte Strings, kann aber eine implizite Typumwandlung vornehmen, so dass Stringliterale zwar den Typ `char*` haben, aber automatisch in `std::string` umgewandelt werden:

```
1 std::string s2 = "abc";
```

Es gibt selten einen Grund, nullterminierte Strings zu verwenden; verwenden Sie `std::string`.

2.9 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 (oder einem anderen Character Encoding). Um diese Zahlen in Berechnungen verwenden zu können, müssen sie erst in `int` oder `double`-Werte umgewandelt werden.

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

```
1  unsigned int asciiStringToNumber(std::string string_value){
2    int  total = 0;
3    int  weight = 1;
4    for(int i=string_value.length()-1; i>=0 ;--i){
5      char c   = string_value[i];  // c is the current digit`s char
6      int  v   = digitToNumber(c); // v is the current digit`s value
7      int part = v * weight;
8  #ifdef DEBUG
9      println("c=", c, " v=", v, " part=", part);
10 #endif
11     total += part;
12     weight *= 10;
13   }
14   return total;
15 }
```

Die Funktion `digitToNumber()` führt die Umwandlung der als Zeichen gegebenen Ziffern in Zahlenwerte anhand der ASCII-Tabelle durch. Zum Beispiel ist der Zahlenwert `48` der ASCII-Wert des Zeichens `0`. Wird 48 von einer solchen Ziffer abgezogen, so erhält man den korrespondierenden Ganzzahlwert.

Da das dezimale Zahlensystem ein Stellenwertsystem ist, muss noch jeder Zifferwert mit der Wertigkeit der jeweiligen Position multipliziert werden (das `*= 10` ergibt die Faktoren 10, 100, 1000, …), bevor alle Zwischenwerte aufaddiert werden (`total += part`).

Hier die `DEBUG`-Ausgabe einer Umwandlung:

```
converting string "1984" to type int.
c = 4   v = +4   part = +4
c = 8   v = +8   part = +80
c = 9   v = +9   part = +900
c = 1   v = +1   part = +1000
result = +1984
```

Oben sind alle einzelnen Rechenschritte vom Zeichen über den Zahlenwert zum Zwischenwert ersichtlich.

Nachfolgend wird die umgekehrte Umwandlung betrachtet. Die Funktion `numberToAsciiString()` wandelt eine als `unsigned int` gegebene Zahl in eine Zeichenkette um, die die zur Zahl korrespondierenden Ziffern enthält.

```
1  std::string numberToAsciiString(unsigned int value){
2    std::string result = "";
3    do {
4      int lastDigit  = value % 10;            // last digit`s int value
5      char c = numberToAsciiDigit(lastDigit); // last digit`s character
6      value =  value / 10;                    // all remaining digits, i.e. shift right
7      result += c;
8  #ifdef DEBUG
9      println("lastDigit=", lastDigit, "  c=", c, "  remainingDigits=", value);
10 #endif
11   } while (value);
12   std::reverse(result.begin(), result.end());
13   return result;
14 }
```

Die Vorgehensweise basiert auf folgender Tatsache: wird bei einer Zahl die ganzzahlige Division mit Rest durchgeführt, so entspricht der Divisionsrest der letzten (rechten) Ziffer. Das Divisionsergebnis entpricht allen restlichen Ziffern (Beispiel: `1337 div 10` ergibt `133` mit dem Rest `7`).

Hier die `DEBUG`-Ausgabe einer Umwandlung:

```
converting int +2052 to type string.
lastDigit = +2   c = 2   remainingDigits = 205
lastDigit = +5   c = 5   remainingDigits = 20
lastDigit = +0   c = 0   remainingDigits = 2
lastDigit = +2   c = 2   remainingDigits = 0
result = 2052
```

2.10 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];
  TypedMemory(uint8_t defaultValue);

  void          putChar(int position, unsigned char c); // position in bytes starting at 0
  unsigned char getChar(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);

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

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:

```
int main() {
  TypedMemory mem(0xcc); // default memory value

  mem.putUInt(0, 0x41312111);

  mem.putChar(4,'C');
  mem.putChar(5,'+');
  mem.putChar(6,'+');
  mem.putChar(7,'!');

  mem.putDouble(0x10, 16.0 /*355.0/113.0*/);

  compund a = {0x48382818, 16.0};
  mem.putAnything(0x20, &a, sizeof(compund));

  RGB rgb = {0x33, 0x44, 0x55};
  mem.putAnything(0x40, &rgb, sizeof(rgb));

  std::cout << mem.hexDump() << std::endl;

  return 0;
}
```

Die Ausgabe ist diese:

```
0000:  41 31 21 11 43 2b 2b 21 cc cc cc cc cc cc cc cc A1!.C++!........
0010:  00 00 00 00 00 00 30 40 cc cc cc cc cc cc cc cc ......0@........
0020:  18 28 38 48 00 00 00 00 00 00 00 00 00 00 30 40 .(8H..........0@
0030:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0040:  33 44 55 cc cc cc cc cc cc cc cc cc cc cc cc cc 3DU.............
0050:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0060:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0070:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
```

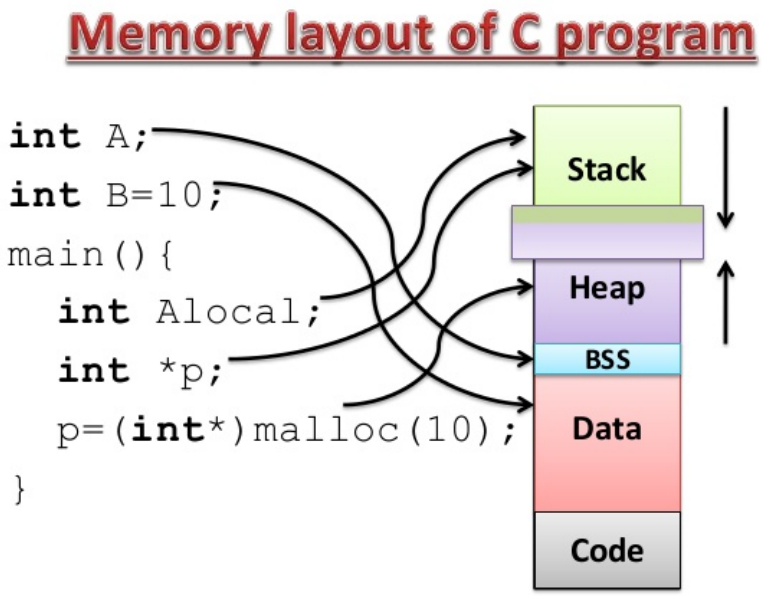
Es ist zu sehen, dass der `unsigned int`-Wert `0x04030201` ab der Speicherstelle `0000` abgelegt ist. Da die Variable `mem` eine lokale Variable ist, enthält sie auch zufällige Werte – daher müssen lokale Variablen immer initialisiert werden.

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 viermal (`byte_a` bis `byte_d`) jeweils genau ein Byte aus `i` herausgeschnitten (mit `&` Bitmaske) und zurechtgerückt (mit `>>` Anzahl) wird. Die vier einzelnen Bytes werden nun nacheinander ab `rawMemory[position]` abgelegt.

2.11 Exkurs: Speicherbereiche zur Laufzeit



Die verschiedenen Speicherbereiche zur Laufzeit mit den jeweils aufgenommenen C-Konstrukten[7].

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

- *Text* oder *Code*: ausführbarer Maschinencode (aus den compilierten C/C++-Dateien und den dazugelinkten Bibliotheken)
- *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.

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

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/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 Sie die Datei `main_03_DATA.cpp` aus dem Quellcode-Verzeichnis (Auszüge davon im unten angegebenen Code) und ergänzen sie folgendes:

- die Funktion `int hexDigitToInt(char)` 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)
- die Funktion `char intToHexDigit(int)` soll eine Zahl zwischen `0` und `15` in die entsprechende Hexadezimalziffer (`'0'` … `'f'`) umwandeln
- die Funktion `PascalString reversed(PascalString)` liefert die übergebene Zeichenkette rückwärts zurück

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 den Compiler für die Übersetzung eine moderne C++-Variante zugrunde legen lassen, sowie einige Warnungen aktivieren, damit Sie auf potentielle Probleme hingewiesen werden. Sie sollten demnach
mit `clang++ -std=c++17 -Wall -Wextra -Wpedantic -O0` übersetzen. Prüfen Sie diesbezüglich die Shell-Variable `CPPFLAGS` in `build.sh`.

```
// 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;
// FIXME: implement!
return value;
}

int hexStringToInt(PascalString hexDigits){
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;
}
```

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

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 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. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

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. Fügen Sie den Rumpf zur Funktion `int hexStringToInt(PascalString)` hinzu, so dass als Zeichenketten gegebene Hexadezimalzahlen in einen korrespondierenden Zahlenwert umgewandelt werden
3. erstellen Sie eine Funktion `intToHexString(int n)`, welche das Argument `n` in einen korrespondierenden hexadezimalen `PascalString` umwandelt
4. erstellen Sie eine Funktion `intToDualString(int n)`, welche das Argument `n` in einen korrespondierenden binären `PascalString` umwandelt (z. B. `5` in `00000101`)
5. (Level C) Erstellen Sie in der Datei `main_colors.cpp` eine Funktion `uint32_t addColors(uint32_t, uint32_t)`, die zwei Farbwerte addiert zurückliefert
6. (Level C) Erstellen Sie in der Datei `main_colors.cpp` eine Funktion `uint8_t luminosity(uint32_t)`, wobei der Rückgabewert mit `0.21 Red + 0.72 Green + 0.07 Blue` berechnet wird
7. (Level C) Erstellen Sie eine Funktion `fromCstring(char *)`, welche es erlaubt, `PascalString`s zu erzeugen
8. (Level C) Fügen Sie die globale Variable `char cStringArea[1024];` hinzu. Erstellen Sie eine Funktion `int to_c_string(PascalString s)` , die eine freie Stelle in diesem Array findet, dort eine nullterminierte Kopie von `s` ablegt und deren Startindex zurückgibt. Lassen Sie die Funktion `to_c_string(PascalString s)` mehrfach aufrufen, um die korrekte Funktion zu prüfen.[8]

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 (Zahlendarstellung im 2er-System) ergibt sich am Ende des folgenden Codes?

```
1 int controlRegister   = 128;      // Bitmuster: 0000 0000 1000 0000
2 controlRegister      |= 64 + 32;
```

[8] Hinweis: Der Einfachheit halber können Sie annehmen, dass Bereiche mit bestimmten Werten (z. B. `255` oder `0`) freie Speicherbereiche sind. Ein möglicher Ansatz ist es, das Problem in kleinen Schritten anzugehen und jeden dieser Schritte in einer eigenen Funktion umzusetzen. Beispielsweise mit `bool enoughCharsFreeAt(int num, int startPos)`, `int findFreeSpaceFrom(int startPos)`, welche von `to_c_string()` verwendet werden.

```
3 controlRegister       ^= 16;
4 controlRegister       &= 128+64;
5 controlRegister       <<= 1;
```

3. Welchen Typ haben die unten angegebenen Ausdrücke?

```
1 1
2 1.0
3 "1.0"
4 1 + 1.0
5 '1'
```

4. Die Funktionen `intToHexString(int n)` und `intToDualString(int n)` aus den Modifikationen haben im Rumpf viele Gemeinsamkeiten. Diskutieren Sie Verbesserungsmöglichkeiten.
5. Wie unterscheiden sich die Werte `"1234"` und `1234` hinsichtlich der Darstellung im Speicher?
6. Warum gibt es so viele verschieden Typen, die Werte ganzer Zahlen aufnehmen können?
7. Welche Schritte sind notwendig, um zwei nullterminierte Zeichenketten an einander zu hängen? Diskutieren Sie mindestens zwei der möglichen Antworten.
8. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- Umrechnung von Zahlensystemen von Arndt Brünner[9]
- Stellenwertsystem von Martin Freidank, Sönke Greve & Felix Graf[10]
- A Tutorial on Data Representation Integers, Floating-point Numbers, and Characters https://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html
- Bit Twiddling Hacks by Sean Eron Anderson https://graphics.stanford.edu/~seander/bithacks.html und deren Implementierung https://github.com/jdbruijn/bit-twiddling-hacks
- Aggregate type in FOLDOC[11]
- 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

[9] https://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm
[10] http://userpages.uni-koblenz.de/~proedler/bac/stellen.php
[11] http://www.dictionary.com/browse/aggregate-type

- Unicode In Python, Completely Demystified: http://farmdev.com/talks/unicode/
- cppreference.com: Fundamental types https://en.cppreference.com/w/cpp/language/types
- cppreference.com: Fixed width integer types https://en.cppreference.com/w/cpp/types/integer
- Diskussion verschiedener 8-Bit Farbpaletten: http://blag.fingswotidun.com/2018/04/giving-kwak8-256-colours.html?m=1

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[12]

[12] https://stratifylabs.co/embedded%20design%20tips/2013/10/18/Tips-RAM-Flash-Usage-in-Embedded-C-Programs/