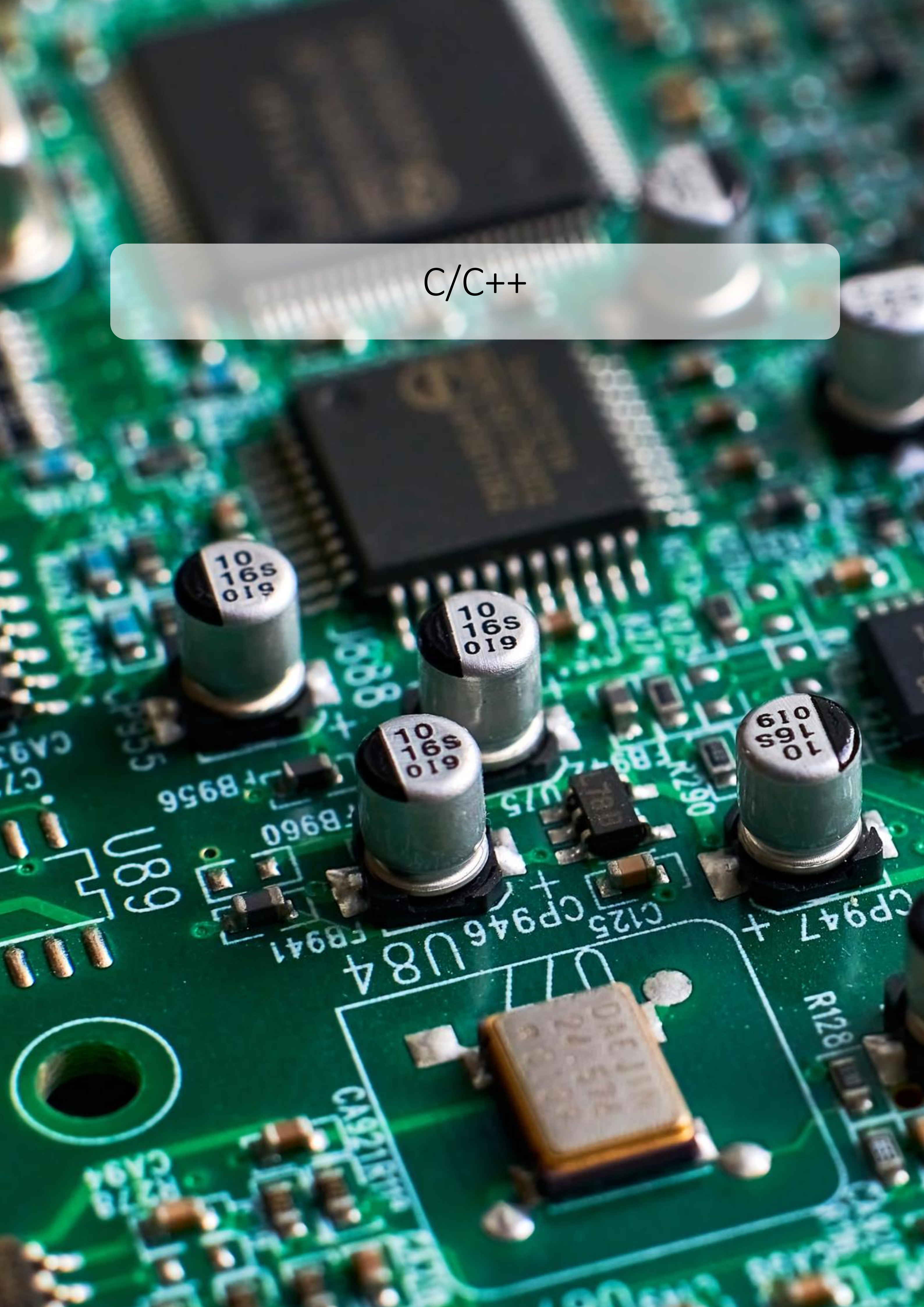

C/C++

Inhaltsverzeichnis

1. Einsatz der Sprache C/C++, Vergleich zu anderen Sprachen 6

1.1. Einführung in die Arbeitsweise des Compilers 6

1.2. Befehle 8

1.3. Endungen 10

2. Grundlegende C Bibliotheken, Ein- und Ausgaben 11

2.1. Datentypen 11

2.2. Standarddatentypen in C 11

2.2.1. Der Datentyp `int` (Integer) 11

2.2.2. Der Datentyp `long` 12

2.2.3. Datentyp `short` 13

2.2.4. Gleitpunkttypen `float` und `double` 14

2.2.5. Datentyp char 14

2.2.6. Datentyp void 14

2.3. Datentypen verwenden 15

3. C Schnellstart 16

4. Bitmanipulation 22

4.1. Bitweises UND 22

4.2. Bitweises ODER 24

4.3. Exklusives ODER (XOR) 25

4.4. Bitweises Komplement (~) 26

4.5. Bitshifting 26

5. Programmabläufe als Skizzen .... Fehler! Textmarke nicht definiert.

5.1. Der Programmablaufplan (PAP) .... ***Fehler! Textmarke nicht definiert.***

5.2. Struktogramm / Nassi-Shneiderman-Diagramm .... ***Fehler! Textmarke nicht definiert.***

6. Logische Ausdrücke und Semantik .... Fehler! Textmarke nicht definiert.

6.1. Semantik (Verhalten) .... ***Fehler! Textmarke nicht definiert.***

6.2. Boolesche Ausdrücke kombinieren .... ***Fehler! Textmarke nicht definiert.***

QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.

6.2.1. Logisches UND ... **Fehler! Textmarke nicht definiert.**
6.2.2. Logisches ODER ... **Fehler! Textmarke nicht definiert.**
6.2.3. Negation ... **Fehler! Textmarke nicht definiert.**

7. Schleifen ... Fehler! Textmarke nicht definiert.

7.1. while-Schleife ... ***Fehler! Textmarke nicht definiert.***
7.2. do-while-Schleife ... ***Fehler! Textmarke nicht definiert.***
7.3. for-Schleife ... ***Fehler! Textmarke nicht definiert.***
7.4. Inkrement / Dekrement ... ***Fehler! Textmarke nicht definiert.***
7.5. Sprünge in Schleifen ... ***Fehler! Textmarke nicht definiert.***
7.6. Endlosschleifen ... ***Fehler! Textmarke nicht definiert.***
7.7. Scopes ... ***Fehler! Textmarke nicht definiert.***
7.8. Konstante Variablen ... ***Fehler! Textmarke nicht definiert.***
7.8.1. Deklaration ... **Fehler! Textmarke nicht definiert.**
7.8.2. Definition ... **Fehler! Textmarke nicht definiert.**
7.8.3. Initialisierung ... **Fehler! Textmarke nicht definiert.**

8. Arrays ... Fehler! Textmarke nicht definiert.

8.1. Einfache Arrays ... ***Fehler! Textmarke nicht definiert.***
8.2. Deklaration eines Arrays ... ***Fehler! Textmarke nicht definiert.***
8.2.1. Array ansprechen ... **Fehler! Textmarke nicht definiert.**
8.2.2. Wertzuweisung ... **Fehler! Textmarke nicht definiert.**
8.2.3. Zugriff ... **Fehler! Textmarke nicht definiert.**
8.2.4. Arrays direkt initialisieren ... **Fehler! Textmarke nicht definiert.**
8.2.5. Char Arrays ... **Fehler! Textmarke nicht definiert.**
8.3. Mehrdimensionale Arrays ... ***Fehler! Textmarke nicht definiert.***
8.4. Mehrdimensionale Arrays direkt initialisieren ... ***Fehler! Textmarke nicht definiert.***

9. Funktionen ... Fehler! Textmarke nicht definiert.

9.1. Was sind Funktionen? ... ***Fehler! Textmarke nicht definiert.***
9.2. Funktionen überladen ... ***Fehler! Textmarke nicht definiert.***
9.3. Funktionen mit Vorgabeargumenten ... ***Fehler! Textmarke nicht definiert.***
9.4. Rekursive Funktionen ... ***Fehler! Textmarke nicht definiert.***
9.5. Arrays an Funktionen übergeben ... ***Fehler! Textmarke nicht definiert.***

10. Lokale Variable .... Fehler! Textmarke nicht definiert.

11. Statische Variablen .... Fehler! Textmarke nicht definiert.

12. Strukturen .... Fehler! Textmarke nicht definiert.

12.1. Anwendung .... ***Fehler! Textmarke nicht definiert.***

12.2. Typedef .... ***Fehler! Textmarke nicht definiert.***

13. Zeiger .... Fehler! Textmarke nicht definiert.

13.1. Einsatz .... ***Fehler! Textmarke nicht definiert.***

13.2. Zeiger (Pointer) deklarieren .... ***Fehler! Textmarke nicht definiert.***

13.3. Dereferenzierung .... ***Fehler! Textmarke nicht definiert.***

13.4. Zeigerarithmetik .... ***Fehler! Textmarke nicht definiert.***

13.5. Speicherzugriffsfehler .... ***Fehler! Textmarke nicht definiert.***

13.6. *Nullpointer* .... ***Fehler! Textmarke nicht definiert.***

13.7. Call by Value / Call by Reference .... ***Fehler! Textmarke nicht definiert.***

13.8. Zeiger auf Zeiger .... ***Fehler! Textmarke nicht definiert.***

13.9. Speicherverwaltung in C .... ***Fehler! Textmarke nicht definiert.***

13.9.1. Dynamische Speicherplatzreservierung mit malloc .... **Fehler! Textmarke nicht definiert.**

13.9.2. Dynamische Speicherplatzreservierung mit calloc .... **Fehler! Textmarke nicht definiert.**

13.9.3. Dynamische Speicherplatzreservierung mit realloc .... **Fehler! Textmarke nicht definiert.**

13.10. Best Practice mit Zeigern .... ***Fehler! Textmarke nicht definiert.***

14. Fehlersuche / Debugging / GDB .... Fehler! Textmarke nicht definiert.

14.1. *Mechanismus* .... ***Fehler! Textmarke nicht definiert.***

14.1.1. Schrittweise Ausführung .... **Fehler! Textmarke nicht definiert.**

15. Zufallszahlen in C .... Fehler! Textmarke nicht definiert.

16. Einführung in C++ und Standardbibliotheken .... Fehler! Textmarke nicht definiert.

16.1. Standardbibliotheken .... ***Fehler! Textmarke nicht definiert.***

16.2. Ein- und Ausgabe auf die Konsole .... ***Fehler! Textmarke nicht definiert.***

16.2.1. Eingabe mit std::getline(…) .... **Fehler! Textmarke nicht definiert.**

16.3. C++-Bibliotheken .... ***Fehler! Textmarke nicht definiert.***

17. OOP mit UML und C++ .... Fehler! Textmarke nicht definiert.

18. Klassen .... Fehler! Textmarke nicht definiert.

18.1. Include-Guards .... ***Fehler! Textmarke nicht definiert.***

18.2. Sichtbarkeiten .... ***Fehler! Textmarke nicht definiert.***

18.3. Der Konstruktor .... ***Fehler! Textmarke nicht definiert.***

18.3.1. Der Standardkonstruktor / Überladener Konstruktor .... **Fehler! Textmarke nicht definiert.**

18.3.2. Konstruktor mit Vorgabeargumenten .... **Fehler! Textmarke nicht definiert.**

18.3.3. Konstruktor mit Initialisierungsliste .... **Fehler! Textmarke nicht definiert.**

18.3.4. Zugriffsspezifizierer (access specifiers) .... **Fehler! Textmarke nicht definiert.**

18.4. Der this-Zeiger .... ***Fehler! Textmarke nicht definiert.***

18.5. Statische Variablen in Klassen .... ***Fehler! Textmarke nicht definiert.***

19. Speicherverwaltung in C++ .... Fehler! Textmarke nicht definiert.

19.1. Dynamischen Speicherplatz verwalten .... ***Fehler! Textmarke nicht definiert.***

19.2. Der Destruktor .... ***Fehler! Textmarke nicht definiert.***

19.3. Virtueller Destruktor .... ***Fehler! Textmarke nicht definiert.***

20. Dateioperationen und Streams .... Fehler! Textmarke nicht definiert.

20.1. Dateien einlesen .... ***Fehler! Textmarke nicht definiert.***

20.2. Daten in Dateien schreiben .... ***Fehler! Textmarke nicht definiert.***

21. Überladen .... Fehler! Textmarke nicht definiert.

21.1.1. Überschreiben von Funktionen (Methoden) .... **Fehler! Textmarke nicht definiert.**

22. friend Funktionen .... Fehler! Textmarke nicht definiert.

23. Makefiles .... Fehler! Textmarke nicht definiert.

24. Vererbungsmuster (Liskovsches Substitutionsprinzip) .... Fehler! Textmarke nicht definiert.

24.1. Liskovsches Substitutionsprinzip .... ***Fehler! Textmarke nicht definiert.***

24.2. Lösungsvorschläge .... ***Fehler! Textmarke nicht definiert.***

24.2.1. Mehrfachvererbung .... **Fehler! Textmarke nicht definiert.**

25. Abstrakte Klassen .... Fehler! Textmarke nicht definiert.

26. Inline Funktionen .... Fehler! Textmarke nicht definiert.

1. Einsatz der Sprache C/C++, Vergleich zu anderen Sprachen

Die Entwicklung von Software für Embedded-Systeme unterscheidet sich erheblich von der, die in anderen Bereichen der IT üblich ist, beispielsweise bei Intel-PCs. So sind bei Embedded-Systemen viele Funktionen stark von der jeweiligen Hardware abhängig, es werden speziell angepasste Betriebssysteme verwendet und beschränkte Ressourcen müssen in der Programmierung berücksichtigt werden.
Hier kommt die Sprache C zum Einsatz. Einer der klassischen Vorteile von C gegenüber anderen Programmiersprachen ist die Möglichkeit jede Adressierung, die hardwaretechnisch möglich ist, in dieser Sprache optimal formulieren zu können. Programmiersprachen, die einem strengeren syntaktischen Konzept folgen erlauben einen solch flexiblen Hardwarezugriff nicht.
Die Syntax von C++ ist eine Obermenge von C und erbt daher die Möglichkeiten ihrer Vorgängerin vollständig. Die Optimierbarkeit von C++ Code und die erzeugte Codegröße sprechen auch für den Einsatz von C++ als objektorientierten Ersatz von C.

1.1. Einführung in die Arbeitsweise des Compilers

Als Übersetzen oder kompilieren bezeichnet man den Vorgang, den als Text geschriebenen Quellcode (engl. *source code*) in eine Sprache zu überführen, die der Computer versteht. Also in Einsen und Nullen. Dies passiert bei C++ in drei Schritten. Um diese zu verstehen, ist zunächst ein Verständnis für die Dateistruktur von C++-Quellcode erforderlich.
Es gibt zwei Kategorien von Quellcode-Dateien in C++:

Quell(code)dateien:

Sie enthalten den eigentlichen Programmcode, die Anweisungen, die das Verhalten des Programms beschreiben. Sie haben die Dateiendung .c bzw. .cpp.

Headerdateien:

Sie enthalten nur Daten, welche die Struktur des Programms beschreiben. Sie haben die Dateiendung .h (vgl. Wikibooks o.J.).
Der Compiler verarbeitet o.g. Dateien nun wie in der folgenden Abbildung dargestellt:

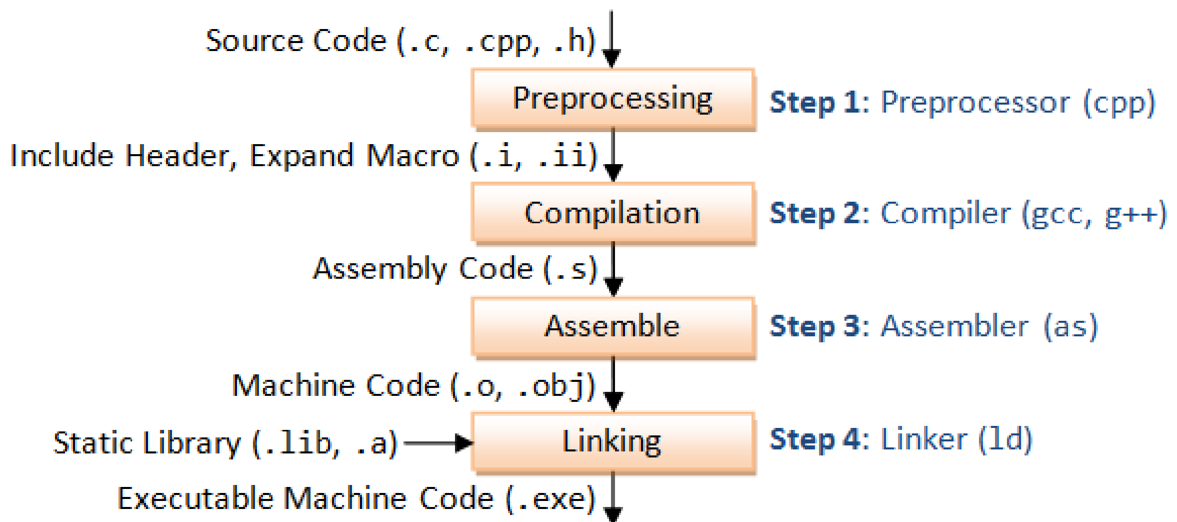

Abb. 1: GCC Compile Prozess. Quelle: Nanyang Technological University (2018)

Schritt 1 „Preprocessing“:

Der Präprozessor verwendet die Quellcode-Datei (Endung .c) als Eingabe und ist für die Verarbeitung der Präprozessoranweisungen verantwortlich. Dazu gehören Include-Dateien, Makros und Anweisungen zur bedingten Kompilierung. Die Ausgabe dieser Stufe ist eine Zwischendatei (Endung .i).

Schritt 2 „Compilation“:

Der Compiler verwendet vorverarbeiteten Code als Eingabe und generiert den entsprechenden Assemblycode. Der Compiler nimmt die .i-Datei als Eingabe und erzeugt eine Assemblycode Datei (Endung .s) als Ausgabe.

Schriftt 3 „Assemble“:

Der Assembler verwendet den Assemblycode (Endung .s) als Eingabe und erzeugt den entsprechenden Maschinencode. Die Ausgabe wird in der Objektdatei (Endung .o) gespeichert.

Schritt 4 „Linking“:

In diesem Schritt des Kompilierungsprozesses wird das Verknüpfen der Objektdateien (engl. Linking) vorgenommen. Der Linker verwendet eine oder mehrere Objektdateien als Eingabe, verknüpft Bibliotheken, löst alle Verweise auf externe Symbole auf, weist Variablen / Funktionen Adressen zu und erstellt das Endergebnis des Kompilierungsprozesses, eine

ausführbare Datei (Endung: Windows .exe, Linux keine). Diese Datei (*Binary*) stellt das ausführbare Programm dar.

Prozess

Für die folgende Darstellung des Kompilier- und Linkingprozesses wird davon ausgegangen, dass drei Quellcodedateien (.c) Teil der Gesamtanwendung sind und zu einem ausführbaren Programm zusammengeführt werden sollen. Es wird deutlich, dass die Objektdateien (.o) ein Zwischenprodukt darstellen, welches vom Linker verarbeitet und zu einer ausführbaren Datei (in Windows zum Beispiel *.exe*) zusammengeführt wird.

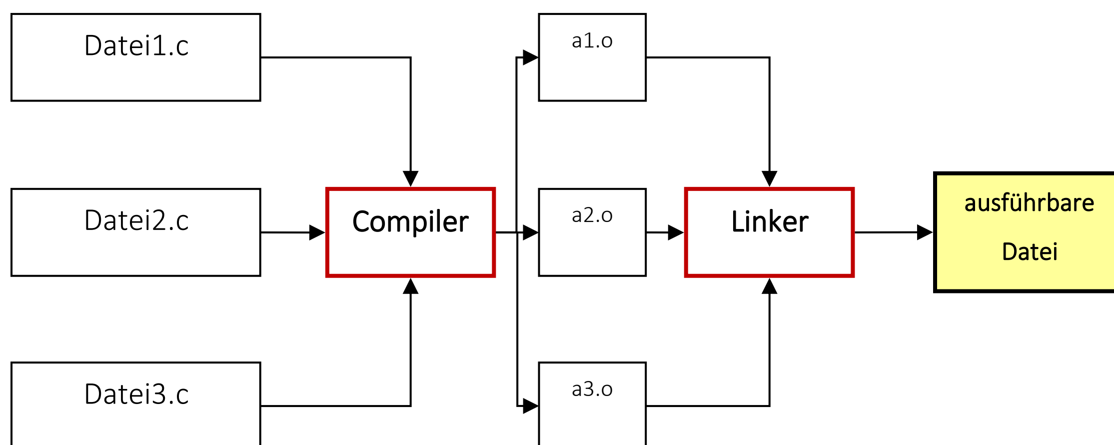

Abb. 2: C Code Kompilier- und Linkingprozess (eigene Abbildung).

1.2. Befehle

Der C-Compiler von GNU, *gcc*, ist einer der vielseitigsten und fortschrittlichsten aller verfügbaren Compiler. Er unterstützt alle modernen C-Standards - etwa ANSI-C - sowie viele Erweiterungen. Zudem kann *gcc* trotzdem zu älteren C-Compilern und älteren Methoden der C-Programmierung kompatibel gemacht werden (vgl. OReilly o.J.).

Der *gcc* Kompiler versteht eine Vielzahl an Kommandozeilenargumenten, sog. *flags*.

Mit dem Befehl **`gcc --version`** bekommt man heraus, welche Compilerversion zum Einsatz kommt. Wird dieser Befehl nicht gefunden ist die Installation zu überprüfen.

Um ein C++ Programm zu kompilieren, verwendet man statt `gcc` den Programmaufruf `g++`.

QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.

Kompilieren eines Programms: `gcc file.c -o HelloWorld`, wobei `file.c` die Quelldatei und *HelloWorld* den Namen des fertigen Programms darstellt.

Weitere nützliche Befehle in nachfolgender Tabelle:

Befehl	Bedeutung
-OX	Weist den Compiler an, den Code weiter zu optimieren; der Grad der Optimierung Platzhalter X kann • 0 (nicht optimiert), • 1 (Größe und Geschwindigkeit), • 2 (Geschwindigkeit), • 3 (Geschwindigkeit, aggressiv), • s (Größe) sein.
-Wall	Weist den Compiler an, alle sinnvollen Warnungen anzuzeigen.
`-Wextra`	Weist den Compiler an, alle Warnungen anzuzeigen.
`-g`	Debugging Informationen hinzufügen.
`-save-temps`	Speichert die temporären Dateien (*intermediate files*), die im Kompilierungsprozess erstellt werden zusätzlich im Verzeichnis der ausführbaren Datei ab.
`-v`	Kann verwendet werden, um während des Kompilierungsprozess Informationen zu jedem Schritt auszugeben.
`-o error.log`	Ausgabe in Datei umleiten.

Tabelle 1: GCC Kommandozeilenanweisungen (eigene Tabelle)

Die `gcc` Manpage[1] (Dokumentationsseiten) kann auf dem Linux System mittels `man gcc` auf der Kommandozeile aufgerufen werden, online findet sie sich unter:

https://linux.die.net/man/1/gcc

Um ein Programm aus mehreren Quelldateien zu kompilieren:

```
gcc -o outputfile file1.c file2.c file3.c.
```

Um mehrere Programme auf einmal mit mehreren Quelldateien zu kompilieren:

```
gcc -c file1.c file2.c file3.c.
```

[1] Erklärung Manpage s.: https://linuxwiki.de/ManPage

QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.

Es ist zu empfehlen, Programme grundsätzlich mit dem Switch –Wall zu kompilieren, um Warnings direkt zu sehen.

```
gcc -Wall myfile.c -o myfile
 myfile.c In function 'main':
 myfile.c:6:6: warning: unused variable 'i'
 myfile.c:7:1: warning: control reaches end of non-void function
```

So werden guter Stil und Codequalität gefördert.

1.3. Endungen

Dateien bzw. Dokumente mit einer solchen Endung können mit dem `gcc` Compiler geöffnet bzw. verarbeitet werden.

`file.c`

C Quellcodedatei, menschenlesbar (Klartext).

Ziel: Präprozessor.

`file.s`

GNU Assembler code, maschinenlesbar.

`file.S`

GNU Assembler code, maschinenlesbar.

Ziel: Präprozessor.

`file.asm`

A68k Assembler code, maschinenlesbar.

`file.o`

COFF Objektdatei (COFF = „Common Object File Format“: Allgemeines Objektdateiformat).

Ziel: Linker.

`file.a`

Statische Bibliothek (Funktionsarchiv).

Ziel: Linker.

2. Grundlegende C Bibliotheken, Ein- und Ausgaben

Bibliotheken beinhalten bereits vorimplementierte Funktionen. Auf diese existierenden Bibliotheksdateien kann, mit dem Zweck, sie im Hauptprogramm zu nutzen, zugegriffen werden.

Mittels des sogenannten `include`-Befehls kann eine Bibliothek eingebunden werden: Er wird immer mit einem hashtag (#) eingeleitet, gefolgt von dem in Klammern gefassten Namen der Bibliotheksdatei. Diese haben die Endung h.

Häufige Verwendung finden die `stdio.h` (Standard-Input-Output-Bibliothek), die eine Sammlung von Funktionen zur Ein- und Ausgabe von Werten enthält, die `ctype.h`, die verschiedene Funktionen, die mit *ASCII*-Zeichen (s. Kapitel Datentypen, Absatz `char`) arbeiten, bereitstellt und die `math.h`, die eine Gruppe mathematischer Funktionen und Konstanten bietet.

2.1. Datentypen

Datentypen sind Arten von Variablen, in denen Daten gespeichert werden können, um zu einem späteren Zeitpunkt wieder darauf zurückzugreifen. Diese Variablen bestehen aus zwei Teilen:

- dem Datentyp, der eine bestimmte Menge Arbeitsspeicher zugewiesen bekommt,
- dem Namen der Variable, mit dem dieser Datentyp im Programm angesprochen werden kann.

Als Basisdatentypen werden einfache, vordefinierte Datentypen bezeichnet. Dies umfasst in der Regel Zahlen (`int`, `short int`, `long int`, `float`, `double` und `long double`), Zeichen (`char`) und den (Nichts-)Typ (`void`).

2.2. Standarddatentypen in C

2.2.1. Der Datentyp `int` (Integer)

Der Datentyp `int` muss, gemäß ANSI C, mindestens eine Größe von zwei Byte aufweisen. Mit diesen zwei Bytes lässt sich ein Zahlenraum von -32768 bis +32767 beschreiben. Mit dem Datentyp `int` lassen sich nur Ganzzahlen darstellen. Die Abkürzung `int` steht für *Integer*. Abhängig vom verwendeten Betriebssystem unterscheidet sich die Größe des Zahlenraums. Vor allem auf Embedded-Systemen spielt dies eine Rolle. Die nachfolgende Tabelle gibt einen

kurzen Überblick über den Datentyp `int` und seinen möglichen Wertebereich auf den verschiedenen Systemen:

System (+ Architektur)	Größe (in Byte)	Wertebereich
Ansi C	2	–32768 +32767
Linux 32-Bit	4	–2147483648 +2147483647
Linux 64-Bit	8	–9.223.372.036.854.755.808 +9.223.372.036.854.755.807
Windows 32-Bit	4	–2147483648 +2147483647
Windows 64-Bit	4	–2147483648 +2147483647
ARM Cortex-M 32-bit	2	–32768 +32767
Atmel 8-bit	4	–2147483648 +2147483647

Tabelle 2:Größe des Int Datentyp auf verschiedenen Systemen (eigene Tabelle, Datenquelle: Kempfer 2018)

Um zu prüfen, welchen Wertebereich der Datentyp `int` auf einem System hat, kann folgender Code mit der Funktion `sizeof()`, verwendet werden:

```
#include <stdio.h>
#include <limits.h>

int main()
{

   printf("int size: %ld byte\n", sizeof(int) );
   printf("from %d to %d\n", INT_MIN, INT_MAX);

   return 0;
}
```

2.2.2. Der Datentyp `long`

Der Datentyp `long` entspricht, wie der Datentyp `int` auch, einer Ganzzahlvariablen. Bei 16-Bit-Systemen hat dieser Typ einen größeren Zahlenbereich und verbraucht somit auch mehr Speicherplatz als der Datentyp `int`.

Der Datentyp `long` hat auf einem 32 Bit System dieselbe Größe und denselben Wertebereich wie der Datentyp `int`. Es gibt ihn aus Kompatibilitätsgründen, damit alte Programme, die für 16-Bit-Rechner geschrieben wurden, auch noch auf einem 32-Bit-Rechner laufen bzw. übersetzt werden können.

```c
#include<stdio.h>

int main()
{
    int var = 5;
    long int longvar = 9999876543L;

    printf("value of var: %d\nsize of var: %lu\n\nvalue of longvar: %ld\nsize of longvar: %lu\n", var, sizeof(var), longvar, sizeof(longvar));

    return 0;
}
```

Dazugehörige Konsolenausgabe:

```
value of var: 5
size of var: 4

value of longvar: 9999876543
size of longvar: 8
```

In diesem Fall sehen wir: Die „normale" int-Variable hat die Größe von 4 Byte. Eine Longvariable die Größe von 8 Byte, also 64 Bit. Um zu kennzeichnen, dass man eine Longvariable meint, wird das L an die entsprechende Zahl gesetzt. Der passende Formatter für die Printausgabe auf der Konsole ist „%ld" („long decimal"). Der Rückgabewert der sizeof()-Funktion ist ein „unsigned long int", der passende Formatter dazu ist „%lu" („long unsigned"). Wenn wir statt „%lu" den Formatter „%d" nutzen, dann erhalten wir zumindest eine Warnung vom Compiler.

2.2.3. Datentyp `short`

Der Datentyp `short` wird verwendet, wenn für eine Variable wenig Speicherplatz benötigt wird oder Speicher gespart werden soll. Er weist eine Größe von 2 Byte und einen Wertebereich von –32768 bis +32767 auf.

```c
#include<stdio.h>

int main()
{
    short int shortInteger = 73;
    printf("value of shortInteger: %hd\nsize of shortInteger: %lu\n", shortInteger, sizeof(shortInteger));


    return 0;
}
```

Konsolenausgabe:

```
value of shortInteger: 73
size of shortInteger: 2
```


QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.

Da das „%s“ als Formatter schon für eine Zeichenkette, also einen string, vergeben ist, wird in diesem Fall der zweite Buchstabe des Wortes „short“ als Formatter genutzt. Der korrekte Formatter für eine „short decimal“ ist folglich „%hd“.

2.2.4. Gleitpunkttypen `float` und `double`

Mit Gleitpunkttypen (*engl. floatingpoint*) wird es möglich, genauere Berechnungen mit Nachkommastellen auszuführen.

`float` weist eine Größe von 4 Byte auf und hat eine Genauigkeit von 6 Stellen.

`double` weist eine Größe von 8 Byte auf (also doppelt so viel, daher der Begriff *double*) und hat eine Genauigkeit von 15 Stellen.

2.2.5. Datentyp char

Der Datentyp `char` dient zur Darstellung von einzelnen Zeichen wie 'a', 'A', 'b', 'B', '5', '7', '§' . Welche Zeichen es gibt, ist mittels der ASCII-Tabelle festgelegt. In der Tabelle ist definiert, dass jedes Zeichen einen Code besitzt. So hat z.B. das Zeichen „A“ den Code 65.

Dezimal	Char	Beschreibung
0	NUL	Null
1	SOH	Start of Header
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission

Tabelle 3: Auszug aus der ASCII Tabelle. Vollständige Tabelle im Anhang (vgl. A1).

2.2.6. Datentyp void

`void` wird bei Funktionen verwendet und ist kein Datentyp im klassischen Sinne. Er findet Einsatz, wenn kein Wert über- oder zurückgegeben werden soll.

Beispiel:

```
void main() {  ...
```

oder

```
int main(void) { ...
```

2.3. Datentypen verwenden

Um Datentypen für Variablen anzuwenden, bedarf es Deklaration, Definition und Initialisierung.

Folgende Schreibweisen sind erlaubt:

```
int wert = 5;
int wert1 = 10, wert2 = 20;

int wert1, wert2 = 33;

int wert1;
int wert2 = wert1 = 10;
```

Bei der Wahl des Bezeichners ist zu beachten:

- Ein Bezeichner darf aus einer Folge von Buchstaben, Dezimalziffern und Unterstrichen bestehen.
- C ist *Case-Sensitive*, d.h. es wird nach Groß- und Kleinschreibung unterschieden.
- Das erste Zeichen darf keine Dezimalzahl sein.
- Reservierte Schlüsselwörter dürfen nicht vergeben werden (eine Variable darf also beispielsweise nicht int, const oder while heißen).

3. C Schnellstart

Erstellen wir unsere erste C-Datei.

Schreibe den folgenden C-Code und speicher die Datei als meinErstesProgramm.c

```
#include <stdio.h>
int main()
{
  printf("Hello World!");
  return 0;
}
```

printf

Ausgaben:

Die `printf`-Funktion nimmt eine Formatzeichenfolge sowie optionale Argumente an und generiert eine formatierte Zeichenfolgensequenz für die Ausgabe. Die Formatzeichenfolge enthält 0 oder mehr *Anweisungen*, die entweder literale Zeichen für die Ausgabe oder codierte *Konvertierungsangaben* sind, die beschreiben, wie ein Argument in der Ausgabe formatiert wird.

Um eine einfache Ausgabe zu machen, reicht folgender Aufruf aus:

```
printf("hallo");
```

Es können jedoch zusätzliche Formatter im Text verwendet werden, die durch Variablen (als Parameter) befüllt werden können.

Hier ein Beispiel:

```
printf("Eins: %d", 1);
```

printf("%d %c", var1 , var2);

Hierbei steht %d für eine Ganzzahl.

Hinter dem %-Zeichen können wir auch zuerst einmal hinschreiben, wie breit das Feld sein soll, in dem die Zahl oder die Zeichenkette ausgegeben werden soll. Dabei wird, wenn nicht anders angegeben, rechtsbündig ausgegeben.

Beispiel:

```
int a = 10;
printf("%5d", a);
```

Dies bedeutet: Wir möchten einen Integer (d steht für „decimal“) in einem Feld der Breite 5 ausgeben. Der Wert 10 wird also auf den letzten beiden Stellen dieses Feldes ausgegeben, also rechtsbündig.

Die Ausgabe wäre also:

   10

Möchten wir die Ausgabe lieber linksbündig haben, dann schreiben wir direkt nach dem %-Zeichen ein -.

Also:

```
int a = 10;
printf("%-5d, Hier geht es weiter!\n", a);
```

Gerne mal ausprobieren. Es ist gut zu sehen, wie die Feldbreite 5 eingehalten wird, der Integer (hier die 10) linksbündig geschrieben ist. Der anschließende Text wird erst nach dem Feld geschrieben.

Außerdem ist es möglich, die Nachkommastellen einer Fließkommazahl anzugeben:

```
    float f = 10.345;
    printf("%.1f", f);
```

Die float-Variable f hat drei Nachkommastellen. Durch „.1“ sorgen wir dafür, dass nur eine Nachkommastelle angegeben wird (mathematisch korrekt auf- oder abgerundet).

Weitere nützliche Formatter sind:

type	*code*	*typical literal*	*sample format strings*	*converted string values for output*
int	d	512	"%14d" "%-14d"	" 512" "512 "
double	f e	1595.1680010754388	"%14.2f" "%.7f" "%14.4e"	" 1595.17" "1595.1680011" " 1.5952e+03"
String	s	"Hello, World"	"%14s" "%-14s" "%-14.5s"	" Hello, World" "Hello, World " "Hello "
boolean	b	true	"%b"	"true"

Der Datentyp *char* steht für "character", also für ein einzelnes Zeichen. In der ASCII-Tabelle wird jedem Zeichen der Tabelle ein Dezimalwert zugeordnet. Somit befindet sich hinter jedem char auch ein Interegerwert. Wir können also sowohl bei einem passenden Integer dank formatter auch das dazugehörige Zeichen ausgeben lassen, als auch zu jeder char-Variablen den dazu passenden Integer.

Das nachfolgende Programm verdeutlicht dies:

```
#include <stdio.h>

int main ()
{
   int ch;
   for( ch = 75 ; ch <= 100; ch++ )
   {
      printf("ASCII value = %d, Character = %c\n", ch , ch );
   }
   return(0);
}
```

scanf

Während printf für die Ausgabe auf die Konsole steht, können wir mit scanf Werte über die Konsole entgegennehmen.

```
#include <stdio.h>

int main()
{
    char name[30];
    printf("Bitte Namen eingeben: ");
```

```c
    scanf("%s", name);
    int alter;
    printf("Bitte Alter eingeben: ");
    scanf("%d", &alter);
    printf("Eingegebener Name: %s \n", name);
    printf("Eingegebenes Alter: %d \n", alter);

    return 0;
}
```

Hier gibt es jedoch eine Besonderheit:

Während bei printf die Variable als solche als Parameter übergeben wird, sieht es bei scanf etwas anders aus. Hier brauchen wir die Adresse. Das wird beim char-Array name nicht so deutlich, aber bei der Abfrage des Alters sehen wir, dass vor der Variable *alter* das &-Zeichen steht, was für den Adressoperator steht. Warum dieser Operator nicht beim array davor steht und wie wir vernünftig mit dem Adressoperator arbeiten, wird etwas später, im Kapitel zu den Zeigern, deutlich. Jetzt ist erst einmal wichtig zu wissen und zu beachten, dass scanf mit der Adresse arbeitet.

Bedingte Ausführung

Eine Verzweigung innerhalb eines Programms wird durch eine Bedingung entschieden. Es wird eine Bedingung formuliert und falls (engl. if) diese zutrifft ein Codeabschnitt ausgeführt. Wenn die Bedingung nicht erfüllt ist, kann man noch eine Alternative (engl. else) setzen.

Syntax

```
if (BEDINGUNG)
   ANWEISUNG
```

Anweisungen steuern den Programmablauf und können schematisch in Programmablaufplänen (abgekürzt: *PAP*) dargestellt werden. Alternative Darstellungsweisen wie das Struktogramm (*gemäß DIN 66261 nach Nassi und Shneiderman ACM SIGPLAN Notices 8, August 1973*) sind ebenfalls zur Darstellung gebräuchlich.

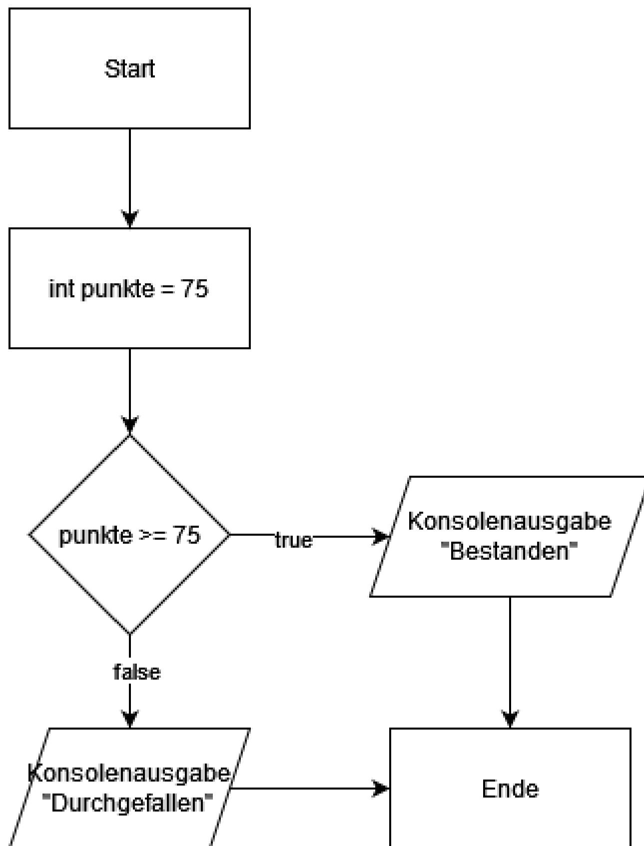

Abb. 3: Programmablaufplan (PAP) Beispiel

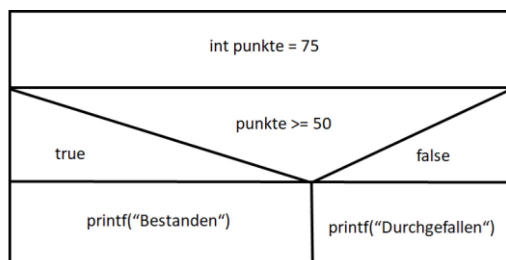

Abb. 4: Struktogramm Beispiel (eigene Abbildung)

Das Diagramm soll einem Prüfling sagen, ob er bestanden hat. Das ist dann der Fall, wenn die Punktzahl (Variable `punkte`) mindestens 50 beträgt. Im Code wird das so gelöst:

```
int punkte = 75;

if (punkte >= 50)
{
   printf("bestanden");
}
```

Die Zeile mit der `printf`-Anweisung wird nur unter der Bedingung ausgeführt, dass der Wert von punkte größer oder gleich 50 ist. Ansonsten wird die Zeile übersprungen.

Eine **Anweisung** ist entweder eine einzelne Code-Zeile (Befehl, Variablenzuweisung) oder ein Code-Block (in geschweiften Klammern).

Eine **Bedingung** ist ein sogenannter **boolescher Ausdruck**. Ein boolescher Ausdruck ist ein Ausdruck, der nach Auswertung immer entweder wahr (*true*) oder falsch (*false*) ist.
Ein weiterer boolescher Ausdruck ist ein **numerischer Vergleich**. Auf beiden Seiten des Vergleichs können beliebig komplexe arithmetische Ausdrücke stehen. Es können Variablen vorkommen. Diese werden bei der Abarbeitung durch ihre aktuellen Werte ersetzt.
Um Vergleiche durchzuführen wird ein Vergleichsoperator benötigt. In C wird mittels doppeltem Gleichheitszeichen (==) auf Gleichheit geprüft. Ungleich wird mittels des Ausrufezeichens vor dem einfachen Gleichheitszeichen (!=) negiert.

4. Bitmanipulation

Mit Hilfe der Bitoperatoren lassen sich Integer-Daten auf der Bit-Ebene manipulieren. Wir kennen bereits die logischen Operatoren für ODER (||) bzw. UND (&&). Diese sind nur deshalb doppelt, weil sie in einfacher Ausführung bereits vergeben sind – und zwar genau für die bitweise Manipulation.

Damit es keine Verwechslung zwischen den logischen und den bitweisen Operatoren gibt, sind die bitweisen Operatoren oft als einzelne Zeichen oder Symbole implementiert, z.B.:

Bitweises ODER (OR): |

Bitweises UND (AND): &

Bitweises exklusives ODER (XOR): ^

Bitweise Negation (NOT): ~

Die logischen Operatoren || und && sind für logische Ausdrücke und Booleans reserviert und können nicht direkt zur bitweisen Manipulation von Integer-Daten verwendet werden. Ihre Verwendung ist auf Booleans beschränkt, während Bitoperatoren auf einer niedrigeren Ebene arbeiten und die einzelnen Bits eines Integer-Werts verarbeiten.

Somit steht | für das bitweise ODER und & für das bitweise UND. Was aber bedeutet das im Klartext?

4.1. Bitweises UND

Wir schauen uns dazu einmal folgende Rechnung an:

```
  0110 1010 (j)
& 0010 1111 (/)
-----------------
  0010 1010 (*)
```

Was soll dies bedeuten?

Das bedeutet: Ein Byte, das die Bitfolge 01101010 hat (das wäre das ASCII-Zeichen ‚j‘) kann durch die Bitweise VerUNDung mit der Bitfolge 00101111 (ASCII-Zeichen ‘/‘) zum Byte mit der Bitfolge 00101010 (ASCII-Zeichen ‘*‘) verändert werden. Wie funktioniert das? Beide Bytes werden bitweise miteinander verglichen. Wir beginnen beim niederwertigsten (rechten) Bit. Im Ergebnis wird nur dann eine 1 gesetzt, wenn beim ersten UND beim zweiten Byte an dieser Stelle eine 1 steht. Das dies in unserem Beispiel nicht der Fall ist, steht im

QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.

niederwertigsten Bit des Ergebnisbytes eine 0. Das zweite Bit (offiziell: Bitnummer 1, weil das niederwertigste Bit das Bit 0 ist) ist sowohl beim ersten als auch beim zweiten Byte „gesetzt“, daher wird es auch im Ergebnisbyte gesetzt. Und so geht’s durchgehend weiter, bis wir das höchstwertige Bit untersucht haben.

Bei aufmerksamem Hinschauen merken wir schnell: Wir hätten das Ursprungsbyte auch mit einer anderen Bitkombination zum Zielbyte manipulieren können (wir hätten statt des ASCII-Zeichens ‘/‘ auch einfach den Punkt ‘.‘ nutzen können – und es gäbe noch einige weitere Beispiele).

Tatsächlich wird die btiweise verUNDung gerne genutzt, um einzelne Bits einfach auszuschalten.

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0

Nehmen wir an, wir wollten das 5. Bit ausschalten, dann könnten wir das vorliegende Byte mit einem Byte verUNDen, das überall gesetzt ist, nur nicht an Stelle 5.

Also:

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0
&	1	1	0	1	1	1	1	1

Als Ergebnis erhalten wir nach zuvor beschriebener Vorgehensweise folgendes Byte:

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0
&	1	1	0	1	1	1	1	1
	0	1	0	0	1	0	1	0

Wir sehen: Es ist fast exakt das gleiche Byte, nur das Bit 5 ist „gekippt“.

Um dies im Programm umzusetzen, bedienen wir uns der binären Schreibweise in C.

Wir schreiben die Bytes als Eins-Null-Kombination, mit einem vorangestellten 0b, also:

```
#include <stdio.h>

int main()
{
    unsigned char c = 0b01101010;
    c = c & 0b11011111;
    printf("ASCII-Dezimalcode: %d\nASCII-Zeichen: %c\n", c, c);

    return 0;
}
```

Wir beschränken uns in diesem und den folgenden Beispielen auf ein Byte, das sich mit einem unsigned char darstellen lässt. Bei einem Integer hätten wir mindestens zwei Byte, in den meisten Fällen wären es 4 Byte, also 32 Bit.

4.2. Bitweises ODER

Während beim bitweisen UND ein Bit nur dann im Ergebnis gesetzt wird, wenn es im ersten UND im zweiten gesetzt ist, sieht es beim bitweisen ODER so aus, dass es genügt, wenn das Bit in mindestens einem der beiden Bytes gesetzt ist. Dies nutzen wir gerne, um Bits gezielt zu setzen.

Schauen wir uns dazu folgendes Byte an:

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0

Nehmen wir an, wir möchten im folgenden Byte das Bit 4 setzen. Wir würden nun eine bitweises ODER machen, in dem wir ein Byte nehmen, das lediglich Bit 4 gesetzt hat:

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0
	0	0	0	1	0	0	0	0
	0	1	1	1	1	0	1	0

Das Ergebnisbyte ist nun fast identisch mit dem Ausgangsbyte, mit dem Unterschied, dass nun Bit 4 ebenfalls gesetzt ist.

Der dazugehörige Code sieht wie folgt aus:

```
#include <stdio.h>

int main()
{
    unsigned char c = 0b01101010;
    c = c | 0b00010000;
    printf("ASCII-Dezimalcode: %d\nASCII-Zeichen: %c\n", c, c);

    return 0;
}
```

4.3. Exklusives ODER (XOR)

Ein Vergleich lässt sich auch so gestalten, dass bitweise geprüft wird, ob nur einer der beiden verglichenen Operanden ein Bit gesetzt hat oder nicht. Im Ergebnis wird nur dann ein Bit gesetzt, wenn sich beide Operanden an entsprechender Stelle unterscheiden:

Bitnummer	7	6	5	4	3	2	1	0
Wert	0	1	1	0	1	0	1	0
^	1	1	0	0	1	1	0	0
	1	0	1	0	0	1	1	0

Wir sehen: Überall, wo sich die obere und die mittlere Zeile unterscheiden, ist im Ergebnis das Bit gesetzt. Haben beide an derselben Stelle eine 0 oder 1, dann steht im Ergebnis eine 0.

Dies wird dazu genutzt, um einzelne Bits zu „kippen". Möchte ich beispielsweise einfach das Bit 5 kippen, dann könnte ich ein XOR mit der Binärdarstellung 0b00100000 machen. Wenn im „Original" das 5. Bit ebenfalls gesetzt war, dann wird es nun zu einer 0. Stand zuvor eine 0 in Bit 5, dann wird es nun zu einer 1.

Da alle anderen Bits im Vergleichsbyte auf 0 stehen, ändert sich an diesen Stellen nichts.

Das wiederum bedeutet: Möchte ich alle Bits kippen, dann würde ich (wir bleiben bei unserem Beispiel mit einem unsigned char, also einem Byte) ein XOR mit 0b11111111 machen.

Bei einem Integer wäre das schon etwas umständlicher, da müsste ich sehr viele Einsen setzen. Das geht aber auch einfacher…

4.4. Bitweises Komplement (~)

Das bitweise Komplement dreht alle Bits um. Aus einer 1 wird eine 0 und andersrum.

Anstatt also „ein XOR mit lauter Einsen zu machen“, bietet es sich an, das bitweise Komplement zu nutzen:

```
#include <stdio.h>

int main()
{
    unsigned char c = 0b10101100;
    unsigned char d = ~c;
    printf("c: %d\t d: %d", c, d);

    return 0;
}
```

Die Ausgabe ergibt für c den Dezimalwert 172 und für d den Dezimalwert 83.

In der Binärschreibweise sind diese beiden einander komplementär, das bedeutet, die Bits sind jeweils umgedreht.

4.5. Bitshifting

Mit Bitshifting können die Bits nach rechts beziehungsweise links verschoben werden.

Nehmen wir an, wir hätten folgende Bitfolge (Dezimal ist dies die Zahl 44):

00101100

Eine Rechtsverschiebung verdeutlichen wir mit dem Operator >>.

Wollen wir jedes Bit um eine Stelle nach rechts verschieben, dann sieht dies so aus:

00101100 >> 1

Dies ist Dezimal die Zahl 22.

Wir können die Bits der ursprünglichen Zahl auch um zwei Stellen nach rechts verschieben:

00101100 >> 2

Das ergibt die Zahl 11.

Würden wir noch eine Stelle nach rechts verschieben, dann würde das Bit 0 (also das niederwertigste Bit, das Bit „ganz rechts“) sozusagen „aus dem Rahmen fallen“. Das wird nicht weiter beachtet, es ist einfach raus. Wir erhalten in diesem Fall dann das Ergebnis 5.

Wir merken: Mit jeder Stelle, die sich die Bits nach rechts verschieben, wird die ursprüngliche Zahl halbiert. Das liegt daran, dass jedes Bit nun nur noch halb soviel wert ist.

Genau so lassen sich die Bits auch nach links verschieben, was konsequenterweise pro Schritt einer Multiplikation mit 2 gleichkommt. Bitshifting wird in der Tat gerne genutzt, um rasch eine Zahl mit einer Potenz der Basis 2 zu multiplizieren:

```
#include <stdio.h>

int main()
{
    int i = 100;
    i = i >> 1;
    printf("%d\n", i);     // Ausgabe: 50

    int j = 10;
    j = j << 3;
    printf("%d\n", j);     // Ausgabe: 80

    return 0;
}
```

Nehmen wir nun an, wir wollten bei der Binärzahl **01101101** das Bit 5 löschen.

Wir könnten es mit folgender Bitmaske verUNDen:

01101101

& 11011111

01001101

Allerdings ist es doch etwas anstrengend, alles auf 1 zu setzen und mittendrin eine 0. Hätten wir es andersrum, also mittendrin eine 1, ansonsten überall nur 0, dann ließe sich Bitshifting total leicht anwenden.

ABER: Auch in diesem Fall können wir uns das Bitshifting zu Nutze machen, und zwar mit folgender Schreibweise:

```
char c = 0b01101101;
c = c & ~(1 << 5);
```

Denn: Die Zahl 1 ist binär 00000001, durch „<< 5“ wird diese 1 auf Bit 5 gesetzt.

Dann sähe es so aus: 00100000. Es soll aber genau andersrum aussehen. Deshalb nutzen wir die Tilde (~), um alle Bits umzudrehen. Durch ~(1 << 5) erhalten wir 11011111.

QUALIDY
SCHULUNGEN, DIE ÜBERZEUGEN.