1 Einführung in die Programmiersprache C

Die Programmiersprache C hat in der Praxis eine sehr große Bedeutung für die hardwarenahe Programmierung. Hardwarenahe Programmierung bedeutet unter anderem, dass die Programmiersprache direkte Zugriffe auf den Arbeitsspeicher des Rechners (Computers), auf dem das Programm läuft, und auch auf Register des Prozessors gestatten muss. Damit ist C ideal für die Systemprogrammierung, bei der betriebssystemnah beziehungsweise hardwarenah programmiert wird. Wegen seiner Hardwarenähe hat C auch für die Programmierung von eingebetteten Systemen – wie beispielsweise den Steuergeräten eines Kraftfahrzeugs – eine herausragende Bedeutung. Die Programmiersprache C liegt schon seit langem in einer von ANSI/ISO standardisierten Form vor.

Das Ziel dieses Kapitels ist es, ein erstes Programmierbeispiel zu zeigen, danach jedoch sogleich auf die Entwicklung und die Eigenschaften der Programmiersprache C einzugehen, die Sprache mit anderen Programmiersprachen zu vergleichen sowie den Stand der Standardisierung von C aufzuzeigen.

Diejenigen Personen, welche so schnell wie möglich mit praktischen Programmierbeispielen beginnen wollen, können die Kapitel über die Entwicklung, andere Sprachen sowie die Standardisierung getrost überspringen und zu einem späteren Zeitpunkt wieder hervorholen.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_1.

J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_1

1.1 Hello World

Programmieren bedeutet, dem Computer Anweisungen zu geben, um ein bestimmtes Problem zu lösen. Diese Anweisungen werden dem Computer mittels einer Programmiersprache verständlich gemacht, was als „**Implementation**“ bezeichnet wird. Implementieren bedeutet Realisieren, Umsetzen, Verwirklichen.

Wie in der Programmiersprache C Anweisungen geschrieben werden, wird hier anhand des weltberühmten „Hello World“-Programms von Kernighan und Ritchie, den Vätern der Programmiersprache C [KR78], gezeigt.

Dieses einfache Programm wird in vielen Programmiersprachen als erstes Beispiel angegeben. Es weist den Computer an, einfach nur den Text `Hello, world!` auf dem Bildschirm auszugeben. In der Programmiersprache C sieht das Programm folgendermaßen aus:

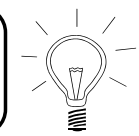
helloWorld.c

```
#include <stdio.h>

int main() {
  printf("Hello, world!");
  return 0;
}
```

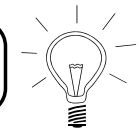
Das „Hello World“-Programm in C besteht aus einer `#include`-Zeile, welche später erklärt wird, sowie einer sogenannten „Funktion“, welche den Namen `main()` trägt und die eigentlichen Anweisungen beinhaltet, welche ausgeführt werden sollen.

In der Programmiersprache C werden Anweisungen in Funktionen geschrieben.



In diesem Buch werden Funktionen, welche im Text auftauchen, immer mit einem leeren Paar runder Klammern versehen, um sie besser kenntlich zu machen. Hier im Beispiel wird die Funktion `main()` betrachtet.

> In C muss jedes Programm eine **`main()`**-Funktion besitzen.



Wenn ein Programm ausgeführt wird, wird mit der Ausführung der `main()`-Funktion begonnen. Die `main()`-Funktion wird auch als Startpunkt eines Programms bezeichnet. Ein Programm darf nur eine einzige `main()`-Funktion besitzen.

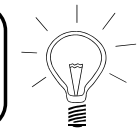
Innerhalb der geschweiften Klammern `{}` werden der Reihe nach die Anweisungen der Funktion `main()` notiert. Unter dem Begriff „**Anweisung**“ wird in C ein mit einem Semikolon `;` abgeschlossener Ausdruck verstanden, welcher verschiedene Sprachelemente eindeutig miteinander verknüpft. Geschweifte Klammern und Anweisungen werden in Kapitel → 2 ausführlicher behandelt.

Während das Programm läuft, werden die Anweisungen nacheinander, also sequenziell hintereinander, ausgeführt. Es ist üblich, mit einer neuen Anweisung in einer neuen Zeile zu beginnen.

Im Falle des „Hello World“-Programms wird als erste Anweisung die Funktion `printf()` aufgerufen. Der Name der Funktion `printf()` steht für „print formatted“. Diese Funktion schreibt `Hello, world!` auf den Bildschirm. Mit der zweiten Anweisung, der `return`-Anweisung, wird die Funktion `main()` auch schon wieder beendet.

Hier wird bereits ein wichtiges Konzept von C sichtbar: Funktionen können wiederum Funktionen aufrufen.

> Alle Programme in C basieren von ihrem Aufbau her komplett auf Funktionen.



1.1.1 Schreiben, Übersetzen und Ausführen eines Programmes

Das oben stehende Programm wird in eine Textdatei geschrieben. Hierfür kann ein einfacher Text-Editor oder eine **Entwicklungsumgebung** (auf Englisch „Integrated Development Environment“, kurz **IDE**) verwendet werden. Die Datei wird unter dem Dateinamen `hello.c` gespeichert.

> Beim Eintippen des Programms muss auf die Groß- und Kleinschreibung geachtet werden, da in C zwischen Groß- und Kleinbuchstaben unterschieden wird.

Um das Programm auszuführen, muss es zunächst mittels eines sogenannten „Compilers“ und „Linkers“ in ein lauffähiges Programm umgewandelt werden. Ein **Compiler** ist ein Programm, welches Programme aus einer Sprache in eine andere Sprache übersetzt. Ein C-Compiler übersetzt beispielsweise ein in C geschriebenes Programm in Anweisungen der sogenannten „**Maschinensprache**“, die der Prozessor eines Computers direkt versteht. Genaueres wird in Kapitel → 5.1 behandelt.

Ein **Linker** verknüpft ein übersetztes Programm mit bereits existierenden übersetzten Programmteilen. Genaueres kann in Kapitel → 5.2 nachgelesen werden. Auch für das „Hello World“-Programm wird der Linker benötigt:

Die Programmiersprache C selbst hat keine eigenen Funktionen für die Ein- und Ausgabe. Hierfür wird die Funktion einer sogenannten „**Bibliothek**“ verwendet. Bibliotheken sind vorgefertigte Sammlungen von Funktionen und weiteren benötigten Programmierelementen. Um Bibliotheksfunktionen nutzen zu können, muss in C eine passende „Schnittstelle“ mittels einer **`#include`**-Angabe eingebunden, sprich bekannt gemacht werden.

Hier im Beispiel des „Hello World“-Programmes wird die Funktion `printf()` benötigt. Damit das Programm `hello.c` somit compiliert und gelinkt werden kann, muss die passende Schnittstelle mittels `#include <stdio.h>` eingebunden werden.

Durch das Einbinden einer Bibliothek werden normalerweise eine Vielzahl an Bibliotheksfunktionen bereitgestellt. Auf die Funktionen der `<stdio.h>`-Bibliothek wird im Detail in Kapitel → 16 eingegangen.

Compilieren und Linken erfolgt bei den heute gängigen Compilern durch einen einzigen Aufruf des Compilers. Wenn Sie mit einer IDE arbeiten, gibt es entsprechende Menübefehle oder anderweitige Bedienelemente, um eine Compilation zu starten. Wenn Sie mit einer **Konsole** (auch „Terminal“, „Eingabeaufforderung“, „Shell“, „Prompt“, „Command Line“ oder „Kommandozeile“ genannt) arbeiten, können Sie mit einem der folgenden Befehle das Programm compilieren. Jede Zeile steht hierbei für einen anderen Compiler.

```
cl hello.c -o hello.exe
gcc hello.c -o hello.exe
clang hello.c -o hello.exe
```

Hier bedeutet `-o hello.exe`, dass das ausführbare Programm `hello.exe` als Output erzeugt werden soll. Unter Unix würde die `.exe` Endung entfallen. Alle Compiler erlauben eine Vielzahl an erweiterten Optionen, auf welche in diesem Buch jedoch nicht eingegangen wird.

Durch Aufruf von `hello.exe` – beziehungsweise unter Unix `hello` – kann das Programm einfach gestartet werden. Wenn Sie mit einer IDE arbeiten, können Sie das Programm wiederum ganz einfach durch Menübefehle oder ein anderes Bedienelement starten.

Nach dem Starten von `hello.exe` wird die gewünschte Ausgabe auf dem Bildschirm angezeigt:

```
Hello, world!
```

Aus der Ausgabe ist ersichtlich, dass die Anführungszeichen " nicht mit ausgegeben werden. Sie dienen nur dazu, den Anfang und das Ende eines sogenannten „**Strings**“ (siehe Kapitel → 6.5.4) zu markieren. Ein String wird im Deutschen auch als **Zeichenkette** bezeichnet und beschreibt wörtlich eine Hintereinanderreihung von einzelnen Zeichen.

1.2 Die Entwicklung von C 

Das Betriebssystem UNIX wurde bei den Bell Laboratorien der Fa. AT&T in den USA entwickelt. Die erste Version von UNIX lief auf einer PDP-7, einem Rechner der Fa. DEC. Diese Version war zunächst in Assembler geschrieben. Assembler ist grundsätzlich die direkte Umsetzung von Maschinensprache in eine für den Menschen verständliche Sprache. Um das System auf einen anderen Rechner portieren zu können, hätte es somit in der für den neuen Rechner gültigen Maschinensprache komplett neu geschrieben werden müssen.

Um das neue Betriebssystem somit einfacher portieren zu können, sollte es in einer sogenannten „höheren Programmiersprache“ neu geschrieben werden. Es sollte aber dennoch sehr schnell sein und Zugriffe auf die Hardware eines Rechners wie den Arbeitsspeicher und Register gestatten.

Eine höhere Programmiersprache zeichnet sich durch eine Expressivität der Sprache aus, welche – mehr als nur einem vorgegebenen Formalismus (**Syntax**) gerecht zu werden – direkt auf die Bedeutung (**Semantik**) des Codes rückschließen lässt. Eine höhere Programmiersprache verwendet Sprachkonstrukte, welche für Menschen intuitiv erscheinen. So steht beispielsweise in C das englische Wort `while` für eine Iteration (Schleife) und das Wort `if` für eine Selektion (Auswahl).

Durch die höhere Expressivität ergibt sich konsequenterweise eine höhere Produktivität und auch weniger Aufwand für die Fehlersuche, da ein solches Programm rascher verstanden wird.

Ebenfalls eine Eigenschaft höherer Programmiersprachen ist die Anwendung erweiterter „Paradigmen“, die über das bloße Abarbeiten von sequenziellen Anweisungen – wie es von Assembler her bekannt ist – hinaus geht.

Durch die gesammelten Erfahrungen mit anderen Programmiersprachen kristallisierten sich zur Zeit der Entwicklung von C zwei Paradigmen heraus: Die sogenannt „Strukturierte Programmierung“ sowie die „Prozedurale Programmierung“. Spätere, noch modernere Sprachen bauten dann auf diesen Konzepten auf und haben „objektorientierte“ und „funktionale“ Paradigmen entwickelt, wie sie heute bekannt sind.

Gesucht war zu Beginn der Entwicklung von C also eine neue Programmiersprache von der Art eines „Super-Assemblers“, der in Form einer höheren Programmiersprache die folgenden Ziele erfüllen sollte:

- Möglichkeiten einer hardwarenahen Programmierung vergleichbar mit Assembler.
- Eine Performance des Laufzeitcodes vergleichbar mit Assembler.
- Eine Unterstützung der Sprachmittel der Strukturierten Programmierung → 1.2.2 sowie der Prozeduralen Programmierung → 4.3.
- Die Möglichkeiten einer maschinenspezifischen Implementierung.

Da eine Programmiersprache aber nicht nur ein Mittel zum Zweck sein sollte, sondern auch ein praktisches und mächtiges Werkzeug, wurden zudem folgende qualitative Eigenschaften für den Charakter der Programmiersprache C definiert [JTC03]:

- Trust the programmer.
- Don’t prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

Eine solche Programmiersprache stand damals noch nicht zur Verfügung. Deshalb entwarf und implementierte Thompson, einer der Väter von UNIX, die Programmiersprache B, beeinflusst von der Programmiersprache BCPL. B musste interpretiert werden und war langsam. Um diese Schwächen zu beseitigen, entwickelte Ritchie 1971/72 die Programmiersprache B zu C weiter. Die Programmiersprache C musste nicht interpretiert werden, sondern für C-Programme konnte von einem Compiler effizienter Code erzeugt werden. Im Jahre 1973 wurde UNIX dann neu in C realisiert, nur rund 1/10 blieb in Assembler geschrieben.

Die Sprache C wurde dann in mehreren Schritten von Kernighan und Ritchie festgelegt. Mit dem Abschluss ihrer Arbeiten erschien 1978 das lange Zeit als „Sprach-Bibel“ betrachtete grundlegende Werk „The C Programming Language“ [KR78]. Mit der Verbreitung von Unix nahm dann C einen rasanten Aufstieg.

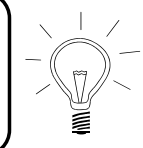
In den folgenden Unterkapiteln werden nun die Eigenschaften von C behandelt:

- Hardwarenahe Programmierung
- Strukturierte und Prozedurale Programmierung
- Abstraktion

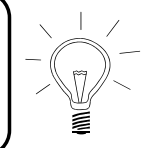
1.2.1 Hardwarenahe Programmierung

C ist eine relativ „maschinennahe“ Sprache. Das bedeutet, dass mit C der Prozessor angewiesen wird, Daten aus dem Arbeitsspeicher zu holen, zu verarbeiten und sie wieder dort zu speichern.

Die Speicherzellen des Arbeitsspeichers sind durchnummeriert. Die Nummern der Speicherzellen werden **Adressen** genannt.



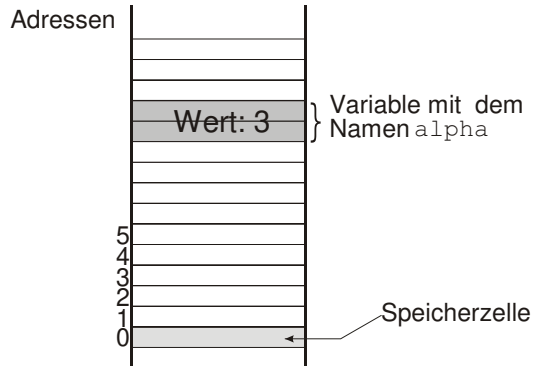
Die Sprache C erlaubt eine **hardwarenahe Programmierung** unter anderem durch direkte Zugriffe auf Adressen im Arbeitsspeicher.



C arbeitet somit mit denselben Objekten wie der Prozessor, nämlich mit Zahlen und Speicheradressen.

In der Regel ist eine Speicherzelle 1 **Byte** groß. Ein Byte stellt in der Regel eine Folge von 8 zusammengehörigen **Bits** dar.

Eine **Variable** ist ein Bereich im Arbeitsspeicher, der über einen Namen angesprochen werden kann. Eine Variable kann auch mehrere Speicherzellen einnehmen. Die Adresse der Variablen ist dabei die Adresse der Speicherzelle, in der die Variable beginnt:



Eine Variable besitzt stets einen sogenannten „**Typ**“, welcher definiert, wieviele Speicherzellen er belegt. Auf das Typkonzept von C wird in Kapitel → 7 genauer eingegangen, hier jedoch eine kurze Auflistung der Basis-Typen, welche in der Programmiersprache C verfügbar sind:

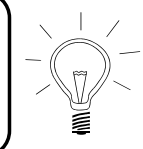
- Integer-Typen
- Gleitpunkt-Typen
- Aufzählungs-Typen
- Strukturen
- Unionen
- Arrays
- Bitfelder
- Adressen

Was scheinbar fehlt sind Zeichen, Strings, sowie boolesche Variablen. Zeichen werden in der Programmiersprache C ebenfalls als Zahlen gespeichert. Strings sind eine Hintereinanderreihung von mehreren solcher Zeichen, was in C mittels eines Arrays passiert (siehe Kapitel → 6.5.4). Boolesche Variablen mit den Wahrheitswerten `false` und `true` gab es im ursprünglichen Sprachkern von C nicht, sondern wurden mit den Zahlen `0` und `1` abgebildet. Für Operationen auf einzelnen Bits innerhalb eines Bytes gibt es in C spezielle Operatoren. Auch die Aufzählungs-Konstanten der sogenannten Aufzählungs-Typen entsprechen in C Integer-Werten.

C-Compiler unterstützen zudem oftmals auch den Zugriff auf Hardware-Register entweder durch spezifische Funktionen, wie beispielsweise durch die Bibliotheksfunktionen `_inp()` und `_outp()` beim Visual C++ Compiler oder durch die Anweisung `asm`. Diese Funktionen wie auch die `asm`-Anweisungen werden jedoch je nach Compiler, Dialekt und System anders verwendet und sind nicht Teil dieses Buches.

1.2.2 Strukturierte und Prozedurale Programmierung

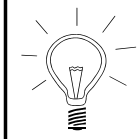
Strukturierte Programmierung erlaubt es, anstelle eines rein sequenziellen Ablaufs von unzähligen Anweisungen einen intuitiven Kontrollfluss zu ermöglichen.



C nutzt hierfür Kontrollstrukturen wie `if`, `while` und `for`. Diese werden im Kapitel → 10 ausführlich behandelt.

Wilde Sprünge im Programm sind bei der Strukturierten Programmierung nicht erlaubt. C enthält aber als hardwarenahe Sprache noch die Sprunganweisung `goto` → 10.11.

Es ist möglich, dass der in C geschriebene **Quellcode** (auch „Quelltext“, oder auf Englisch **source code** genannt) eines Programms aus mehreren Dateien bestehen kann. Jede dieser Dateien kann getrennt in Maschinensprache übersetzt werden. Dabei ist Maschinensprache eine Sprache, die ein bestimmter Prozessor versteht.

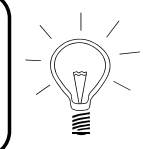


Der Aufbau eines Programms aus getrennt compilierbaren Dateien, den Modulen, wird in Kapitel → 15 unter dem Thema „Speicherklassen“ behandelt.

Die separate Compilierung der verschiedenen Module bietet große Vorteile, zum einen für die Verwaltung der Programme, zum anderen für den Vorgang des Compilierens:

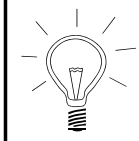
Bei Änderungen im Code muss gegebenfalls nur eine einzige Datei als kleiner Baustein des Programms geändert werden. Die anderen Dateien bleiben in diesem Fall stabil. Bei komplexen Programmen kann der Compilierlauf des gesamten Programms wesentlich länger als der Compilierlauf einzelner Dateien dauern. Es ist also günstig, wenn nur einzelne Dateien neu compiliert werden müssen.

Prozedurale Programmierung ermöglicht es, Teile des Codes in kleinere Einheiten zu verpacken. In C werden hierfür die sogenannten „**Funktionen**“ verwendet.



Durch Hinzufügen von Parametern zu solchen Funktionen können beliebige „komplexe Anweisungen“ programmiert werden, welche sodann an beliebiger Stelle aufgerufen werden können. Durch die Benennung solcher Funktionen mit eingängigen Namen wird zudem der Code lesbarer.

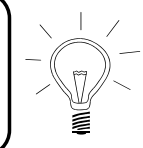
Eine prozedurale Sprache wie C stellt Techniken für die Definition von Funktionen und deren Parametrisierung sowie für den Aufruf von Funktionen, die Argumentübergabe und die Rückgabe von Ergebnissen bereit.



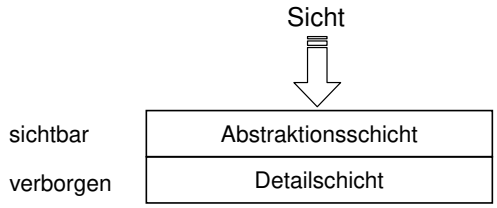
Die Konzepte der Prozeduralen Programmierung werden im Detail in Kapitel → 4.3 abgehandelt.

1.3 Abstraktion

Abstraktion bedeutet grundsätzlich, dass bei komplexen Sachverhalten unwesentliche Dinge unsichtbar gemacht werden, sodass nur das Wesentliche sichtbar bleibt.



Das symbolisiert das folgende Bild:



Durch die allmähliche Entwicklung weg von der rein elektronischen Datenverarbeitung (EDV) hin zur Informationsverarbeitung (auf Englisch „Information Technology“, oder kurz IT) mussten somit Abstraktionskonzepte auch auf Proramm-iersprachen angewendet werden.

Bei der Entwicklung der Programmiersprachen kann im Nachhinein festgestellt werden, dass es drei große Fortschritte im Abstraktionsgrad gab [Bar83]:

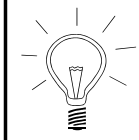
- Abstraktion bei Ausdrücken
- Abstraktion bei Kontrollstrukturen
- Abstraktion von Daten

1.3.1 Abstraktion bei Ausdrücken

Allgemein ist ein **Ausdruck** eine Verknüpfung von Operanden, Operatoren und runden Klammern wie beispielsweise `3 * (4 + 7)`.

Den ersten Fortschritt in der Abstraktion von Audrücken brachte die Programmiersprache FORTRAN (**FOR**mula **TRAN**slation). Die Programmiersprache C hat diese Eigenschaften übernommen.

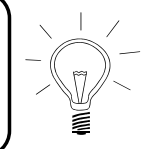
> Während in Assembler für komplexere Ausdrücke mehrere Instruktionen geschrieben werden müssen, welche zudem direkt auf die Maschinenregister zugreifen, können in C direkt mathematische Ausdrücke wie beispielsweise `3 * (4 + 7)` hingeschrieben werden.



Die Umsetzung auf die Maschinenregister wird durch den Compiler vorgenommen und bleibt vollständig verborgen.

1.3.2 Abstraktion bei Kontrollstrukturen

> Unter einer **Kontrollstruktur** wird die Beeinflussung der Abarbeitungsreihenfolge von Anweisungen aufgrund von Bedingungen verstanden.



Unter Assembler und selbst FORTRAN mussten noch manuell Sprungmarken definiert werden, an welche der Prozessor springen soll:

```
      IF (A-B) 100, 200, 300
100   ...
      GOTO 400
200   ...
      GOTO 400
300   ...
400   ...
```

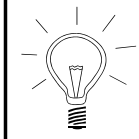
Mit der Programmiersprache ALGOL 60 (**ALGO**rithmic **L**anguage **60**) wurde zum ersten Mal die Iteration und Selektion in abstrakter Form zur Verfügung gestellt, ohne dass einzelne Punkte im Programmablauf mit Marken benannt und dorthin gesprungen werden musste. Die Programmiersprache C hat diese Eigenschaft übernommen, sodass derselbe Ausdruck in C folgendermaßen aussieht:

```
if (A - B < 0) {
  ...
} else if (A - B == 0) {
  ...
} else {
  ...
}
```

1.3.3 Datenabstraktion

Ein großer Fortschritt in der Geschichte der Programmiersprachen war das Konzept der Datenabstraktion.

> Mit dem Konzept der **Datenabstraktion** wurde das Ziel verfolgt, die Einzelheiten der Datendarstellung von den Beschreibungen der Operationen auf den Daten zu trennen, um eine gesteigerte Übertragbarkeit und Wartbarkeit sowie eine höhere Sicherheit zu erreichen. Bei diesem Konzept bleibt die Darstellung der Daten als Bits und Bytes verborgen, bekannt sind nur die Operationen zum Zugriff und zur Manipulation der Daten.



In der Programmiersprache C wird diese Abstraktion durch die Verwendung eines klaren Typsystems unterstützt. Jede Variable und selbst jede Funktion besitzt einen eindeutigen, statischen Typ, welcher vom Compiler ermittelt wird. Eine **Signatur** gibt einer Funktion einen Namen und definiert die Übergabeparameter. Welche Operationen mit welchen Daten und welchen Funktionen möglich sind, ist somit durch die Vergabe der Typen definiert. Weicht irgend eine Benutzung einer Variablen oder Funktion von der durch die Sprache vorgegebenen Typregeln ab, so meldet der Compiler einen Fehler.

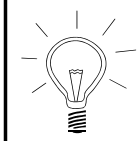
Die Nutzung ausführlicher und verständlicher Bezeichner (Namen) von zu verarbeitenden Daten und aufzurufenden Funktionen lässt zusätzlich auf deren Bedeutung schließen. So muss man sich in C nicht mehr um die korrekte Speicherung einer Variablen an den entsprechenden Speicherzellen kümmern, sondern kann einfach sie mit ihrem Namen ansprechen oder etwas ihr zuweisen.

Eine Variable mit dem Namen `gewicht` beispielsweise lässt bereits erahnen, dass in dieser Variablen ein Gewicht abgespeichert wird. In der modernen Programmierung sind nicht-aussagekräftige Namen unerwünscht. Eine Ausnahme von dieser Regelung bilden Bezeichner wie `i` und `j`, die häufig als einfache Zählvariablen für einen Schleifenindex verwendet werden.

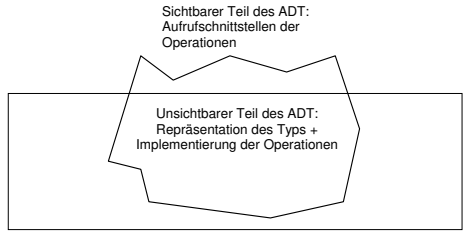
Auch ermöglicht es C, Daten mittels Strukturen zu größeren Informationseinheiten zu verkapseln, um sie so geordnet ansprechen zu können. Die Verwendung des strukturierten Typs wird in Kapitel → 13.1 besprochen.

Das grundlegende Konzept der Datenabstraktion wird durch den Begriff des sogenannten „abstrakten Datentyps“ verkörpert.

> Ein abstrakter Datentyp (ADT) basiert auf der Formulierung abstrakter Operationen auf abstrakt beschriebenen Daten. Um die erlaubten Operationen zu spezifizieren, benötigen sie jeweils eine Signatur und eine Semantik (Bedeutung).



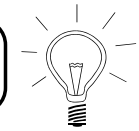
Bertrand Meyer [Mey97] symbolisiert einen abstrakten Datentyp durch einen Eisberg, von dem nur der Teil über Wasser – sprich die Aufrufschnittstellen der Operationen – sichtbar ist. „Unter Wasser“ und damit im Verborgenen liegen die Repräsentation der Daten und die Implementierung der Operationen:



C unterstützt dieses Konzept nicht vollumfänglich. Erst das Klassenkonzept der objektorientierten Programmiersprachen wie beispielsweise C++ ermöglichte es, dass Daten und die Operationen, die mit diesen Daten arbeiten, zu einem gemeinsamen Datentyp – der Klasse – zusammengefasst werden können.

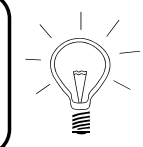
1.4 Vorgänger und Nachfolger von C

C wird zu den **imperativen Sprachen** gezählt.



Imperative Sprachen sind geprägt durch die von-Neumann-Architektur eines Rechners, bei der Befehle im Speicher die Daten im gleichen Speicher bearbeiten.

Bei imperativen Sprachen besteht ein **Programm** aus Variablen, die Speicherstellen darstellen, und einer Folge von Befehlen, die Daten verarbeiten.



Der **Algorithmus** ist bei den imperativen Programmiersprachen der zentrale Ansatzpunkt. Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems. Die Verarbeitungsschritte und ihre Reihenfolge müssen also im Detail festgelegt werden, um zu einem gewünschten Ergebnis zu gelangen.

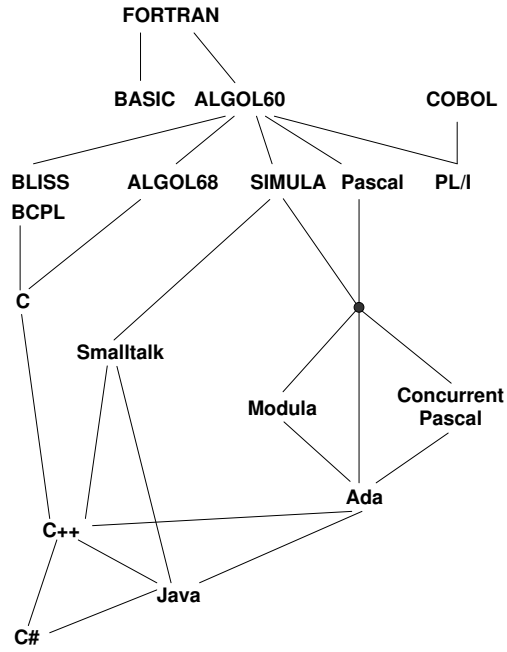
Weitere Beispiele für imperative Sprachen neben C sind FORTRAN, COBOL und Pascal, aber auch Java und C#.

Im Gegensatz dazu stehen die „deklarativen Sprachen“. Bei ihnen werden nicht mehr die Verarbeitungsschritte angegeben, sondern das gewünschte Ergebnis wird direkt beschrieben, also „deklariert“. Ein Übersetzer beziehungsweise Interpreter muss daraus die Verarbeitungsschritte ableiten. Zu den deklarativen Sprachen zählen sogenannte „funktionale Sprachen“ wie zum Beispiel LISP, sogenannte „logikbasierte Sprachen“ wie PROLOG und sogenannte „regelbasierte Sprachen“ wie OPS5.

Bei den imperativen Sprachen werden folgende Sprachen unterschieden:

- Maschinenorientierten Sprachen wie Assembler.
- Prozeduralen Sprachen wie FORTRAN, ALGOL, Pascal, C.
- Objektorientierten Sprachen wie Smalltalk, EIFFEL, C++, Java und C#.

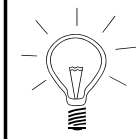
Das folgende Bild zeigt einen Stammbaum imperativer Programmiersprachen:



Anzumerken ist, dass C zwar schon etwas älter ist, dass aber viele moderne objektorientierten Programmiersprachen wie C++, Java und C# in ihrer Syntax auf C basieren, wobei allerdings das Konzept der Objektorientierung eine neue Dimension darstellt.

1.4.1 Die Sprache C++

Während C als „Super-Assembler“ für hardwarenahe Software entwickelt wurde, liegt die Zielrichtung bei der Entwicklung von **C++** darin, neue Sprachmittel wie beispielsweise Klassen bereitzustellen, um Anwendungsprobleme noch intuitiver zu formulieren.



C++ baut auf dem Funktionsumfang von C auf und erweitert diesen um das Paradigma der „objektorientierten Programmierung“. Die grundlegende Idee hierbei ist, dass eigene Datentypen definiert werden können, für welche genau festgelegt werden kann, welche Operationen zulässig sind.

C++ gilt als der direkte Nachkomme von C und hat sich über die Jahre stark weiterentwickelt. Mittlerweile sind auch Konzepte der funktionalen Programmierung sowie moderner Design-Paradigmen eingeflossen.

Die fest verankerte Rückwärtskompatibilität zu C macht C++ zu einer sehr mächtigen Partnersprache von C: Während C beispielsweise für die Ansteuerung von Hardware verwendet wird, kann C++ die Präsentation mittels eines User Interfaces (GUI) übernehmen. In C geschriebene Funktionalität kann direkt aus einem Programm geschrieben in C++ aufgerufen werden. Mit anderen Sprachen ist diese Verknüpfung nicht so einfach möglich.

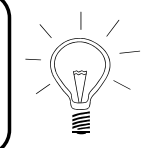
1.5 Standardisierung von C

Wie bei UNIX selbst, so entwickelten sich anfänglich auch bei C-Compilern viele verschiedenartige Dialekte, was zu einer erheblichen Einschränkung der Portabilität von C-Programmen führte.

Im Jahre 1989 wurde C durch das ANSI-Komitee X3J11 normiert mit dem Ziel, die Portabilität von C zu ermöglichen. Programme, die nach **ANSI-C** (X3.159-1989) geschrieben wurden, konnten von jedem Compiler auf jedem Rechner compiliert werden, vorausgesetzt, der Compiler war ein ANSI-C-Compiler oder enthielt ANSI-C als Teilmenge.

Der ANSI-Standard normierte nicht nur die Sprache C, sondern auch die Standardbibliotheken, ohne die C nicht auskommt, da C selbst – wie bereits erwähnt – beispielsweise keine Einrichtungen für die Ein- und Ausgabe hat.

Bibliotheken sind im Unterschied zu Programmen nicht selbstständig ablauffähig. Bibliotheken enthalten Hilfsmodule, welche von Programmen angefordert werden können.



In der Folgezeit wurde der Standard ANSI X3.159-1989 mit kleinen Änderungen von der ISO als internationaler Standard ISO/IEC 9899 [C90] übernommen. Diese Sprachversion wird nach dem Veröffentlichungsjahr **C90** genannt. C90 wird bis heute von fast allen Compilern unterstützt und ist in der Industrie immer noch verbreitet.

1995 erschien die erste Erweiterung zur C-Norm. Der unter dem Namen ISO/IEC 9899/AMD1:1995 (oder auch **C95** genannt) veröffentlichte Standard enthielt neben Fehlerbehebungen nur kleine Änderungen am Sprachumfang.

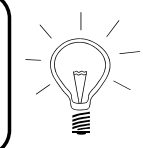
In einer weiteren Überarbeitung wurden auch Sprachkonzepte aus C++ übernommen. Der Standard ISO/IEC 9899 [C99] erschien 1999 und wird als **C99** bezeichnet.

C11 ist der informelle Name des im Dezember 2011 veröffentlichten ISO-Standards ISO/IEC 9899:2011 [C11] für die Programmiersprache C. Er ersetzt den vorigen Standard C99.

Der Standard **C17** gilt als ein Zwischenstandard, welcher kleinere Fehler des C11 Standards korrigierte. Da er selbst jedoch keine neuen Spracheigenschaften einführte, wird umgangssprachlich immer noch C11 als der aktuelle Standard bezeichnet.

Zum Zeitpunkt des Schreibens dieses Buches ist ein neuer Standard in Bearbeitung, welcher voraussichtlich mit **C23** benannt werden wird. Die wichtigsten neuen Spracheigenschaften werden in diesem Buch bereits beschrieben.

Ein neuer C-Standard ersetzt immer seinen Vorgänger. Es kann zur selben Zeit immer nur einen einzigen gültigen Standard geben.



Die grundsätzlichen Unterschiede zwischen den Standards werden in diesem Buch nicht beschrieben. Stattdessen werden nur vereinzelt Besonderheiten erwähnt werden, die für das Studium der Sprache interessant sind.