

## 5 Programmerzeugung und -ausführung



Der Quellcode eines Programms wird mit einem **Editor**, einem Werkzeug zur Erstellung von Texten, geschrieben und auf der Festplatte des Rechners unter einem Dateinamen als Datei abgespeichert.

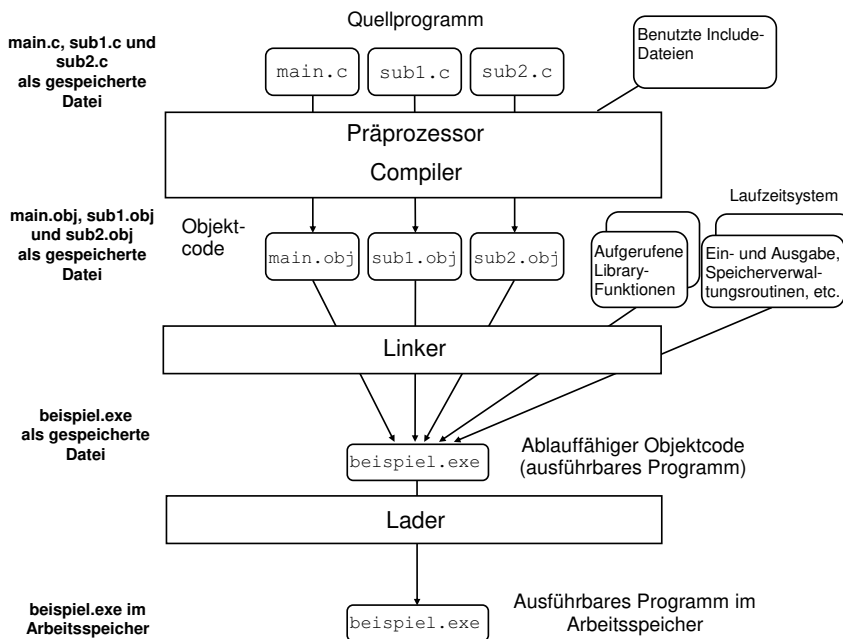


Da eine solche Datei Quellcode enthält, wird sie auch als **Quelldatei** (auf Englisch „**source file**“) bezeichnet.

Einfache Programme bestehen aus einer einzigen Quelldatei, komplexe aus mehreren Quelldateien.



Das folgende Bild zeigt einen Überblick über den Ablauf der Tätigkeiten, die durchzuführen sind, um ein C-Programm zu erzeugen und es auszuführen:



In diesem Kapitel werden alle Stationen dieses Diagramms ausführlich erläutert. Hier jedoch vorab die Kurzzusammenfassung:

Das gezeigte Beispielprogramm besteht aus drei separat compilierbaren Quelldatei `main.c`, `sub1.c` und `sub2.c`. Beim Compiler-Aufruf läuft in C vor dem eigentlichen Compiler als erstes der Präprozessor durch das Programm.

Zuerst muss der **Präprozessor** in den Quelldateien die Präprozessor-Anweisungen auflösen, bevor der Compiler die Dateien compilieren kann.



Durch Compilieren erhält man aus einer Quelldatei eine **Objektdatei**.



Das Auflösen der Präprozessor-Anweisungen geschieht bei den meisten Compilern automatisch, so dass kein Zwischenprodukt entsteht, sondern dass eine Objektdatei in einem einzigen Durchlauf aus einer Quelldatei resultiert.

Zu jeder Quelldatei wird durch den Compiler eine entsprechende Objektdatei erzeugt. Unter Windows wird normalerweise für Objekt-Dateien die Extension `.obj` verwendet, unter UNIX/Linux normalerweise die Extension `.o`.

Der Linker erzeugt aus den Objektdateien ein ausführbares **Programm**.



Ein ausführbares Programm hat unter Windows per Konvention die Extension `.exe`.

Ein ausführbares Programm kann mit Hilfe eines Laders beliebig oft zur Ausführung gebracht werden.



Im Folgenden wird die Funktionalität von Compilern, Linkern und Ladern genauer besprochen. Ferner werden Debugger und integrierte Entwicklungsumgebungen vorgestellt.

## 5.1 Compiler

Die Aufgabe eines **Compilers** (Übersetzers) für die Programmiersprache C ist, den Text (Quellcode) eines C-Programms in Maschinensprache zu wandeln.



**Maschinensprache** (auch „**Maschinencode**“ genannt) ist prozessorspezifischer Binärcode, welcher von einem speziellen Prozessor ohne weitere Übersetzung direkt verstanden und verarbeitet werden kann. Ein vollständig für den Prozessor übersetzter Programmablauf besteht somit grundsätzlich nur aus einer Folge von Nullen und Einsen und wird deswegen auch als das sogenannte „**binary**“ bezeichnet.



Da Maschinensprache die tiefstmögliche Art der Kommunikation mit dem Prozessor darstellt, wird eine Sprache wie C, die vom speziellen Prozessor abstrahiert, als „**höhere Programmiersprache**“ bezeichnet. Während die ersten Sprachen und Compiler in den 50er und 60er Jahren noch heuristisch entwickelt wurden, wurde die Spezifikation von Sprachen und der Bau von Compilern zunehmend formalisiert.

ALGOL 60 war die erste Sprache, deren Syntax formal definiert wurde. Als **Syntax** bezeichnet man die Schreibweise, also die Form und Struktur einer Sprache, im Deutschen etwa die Regeln für den Satzbau. Diese Definition der Syntax wurde mit Hilfe der sogenannten „Backus-Naur-Form“ (BNF) verfasst [NBB<sup>+</sup>63]. Die Backus-Naur-Form ist dabei eine sogenannte „Metasprache“, also eine Sprache, welche eine Sprache beschreibt.

Unabhängig von der Art der höheren Programmiersprache kann ein Compiler – bei Einhaltung bestimmter Regeln bei der Definition einer Sprache – grob in folgende 5 Bearbeitungsschritte gegliedert werden:

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse
- Optimierungen
- Codeerzeugung

Die Zwischenergebnisse dieser Schritte werden während des Compilierens in Form von Zwischensprachen und ergänzenden Tabelleneinträgen intern gespeichert und an den nächsten Schritt weitergegeben.

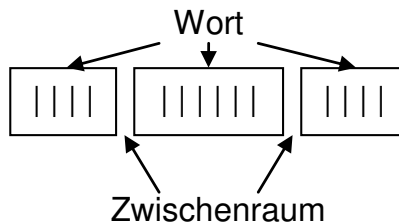
### 5.1.1 Lexikalische Analyse

Bei der lexikalischen Analyse wird versucht, in der Folge der Zeichen eines Quellcodes Einheiten der Sprache zu erkennen. Die lexikalische Analyse wird auf Englisch „Lexer“ genannt, auf Deutsch wird sie auch als „Scanner“ oder „Symbolentschlüsselung“ bezeichnet.



Dabei werden die kleinsten Einheiten einer Sprache ermittelt, die für sich selbst eine Bedeutung besitzen. Eine solche Einheit wird als ein „Wort“ bezeichnet und kann beispielsweise ein Name, ein Schlüsselwort, ein Operator, eine Zahl, etc. sein.

Folgendes Bild demonstriert das Erkennen von Wörtern. Die Striche sollen die einzelnen Zeichen einer Quelldatei darstellen.



Zwischenräume und Kommentare dienen dem Compiler dazu, zu erkennen, an welcher Stelle ein Wort zu Ende ist. Ansonsten haben sie keine Bedeutung für den Compiler und werden überlesen.

### 5.1.2 Syntaxanalyse

Für alle modernen Sprachen existiert ein Regelwerk, die sogenannte Grammatik. Die Grammatik einer Sprache legt formal die zulässigen Folgen von Symbolen (Wörtern) fest.

Im Rahmen der **Syntaxanalyse** wird geprüft, ob die bei der lexikalischen Analyse ermittelte Symbolfolge eines zu übersetzenden Programms zu der Menge der korrekten Symbolfolgen gehört.



Der Teil des Compilers, der die Syntaxanalyse durchführt, wird auch als „Parser“ bezeichnet.

### 5.1.3 Semantische Analyse



Die **semantische Analyse** versucht, die Bedeutung der durch die lexikalische Analyse erkannten Wörter herauszufinden und hält diese meist in Form eines Zwischencodes fest. Ein Zwischencode ist nicht mit der Maschinensprache einer realen Maschine vergleichbar, sondern auf einer relativ hohen Ebene angesiedelt. Er ist für eine hypothetische Maschine gedacht und dient einzig dazu, die gefundene Bedeutung eines Programms für die nachfolgenden Phasen eines Übersetzers festzuhalten.

Die Bedeutung in einem Programm bezieht sich im Wesentlichen auf dort vorkommende Namen. Also muss die semantische Analyse herausfinden, was ein Name, der im Programm vorkommt, bedeutet. Grundlage hierfür sind die Sichtbarkeits-, Gültigkeits- und Typregeln einer Sprache. Mehr dazu kann in Kapitel [→ 11.2](#) nachgelesen werden.

Jeder Name wird mit einer Bedeutung versehen, also an eine Deklaration des Namens im Programm gebunden. Eine Deklaration gibt im Programm den Typ eines Namens bekannt, beispielsweise ob es sich um eine Variable oder eine Funktion handelt, einen Integer- oder um einen Gleitpunkt-Typ.

Neben der Überprüfung, ob Namen im Rahmen ihrer Gültigkeitsbereiche verwendet werden, spielt die Überprüfung von Typverträglichkeiten bei Ausdrücken bei der semantischen Analyse eine Hauptrolle.



Ein wesentlicher Anteil der semantischen Analyse befasst sich also mit der Erkennung von Programmfehlern, die durch die Syntaxanalyse nicht erkannt werden konnten, wie beispielsweise die Addition von zwei Werten mit nicht verträglichem Typ. Nicht alle semantischen Regeln einer Programmiersprache können jedoch durch den Übersetzer geprüft werden.

Man unterscheidet zwischen der statischen Semantik (durch den Übersetzer prüfbar) und der dynamischen Semantik (erst zur Laufzeit eines Programms prüfbar). Die Prüfungen der dynamischen Semantik sind üblicherweise im sogenannten Laufzeitsystem realisiert (siehe Kapitel [→ 5.1.6](#) ).



### 5.1.4 Optimierungen

In der **Optimierungsphase** wird versucht, den Code, so wie er jetzt vorliegt, zu verbessern, sowohl bezüglich des Speicherplatzverbrauchs als auch bezüglich der benötigten Rechenzeit.



Ein Compiler kann beispielsweise nicht benutzten Code ignorieren, sich repetierende Ausdrücke kombinieren oder Prüfungen eliminieren, wenn der zu testende Wert bereits zur Compilerzeit feststeht. Es gibt auch noch kompliziertere Optimierungen, die einen hohen Aufwand während der Übersetzung erfordern. Bei vielen Compilern können all diese Optimierungen beliebig konfiguriert werden.

### 5.1.5 Codeerzeugung

Während die lexikalische Analyse, Syntaxanalyse und semantische Analyse sich nur mit der Analyse des zu übersetzenden Quellcodes befassen, kommen bei der Codegenerierung die Rechnereigenschaften, nämlich die zur Verfügung stehende Maschinensprache und die Eigenschaften des Betriebssystems, ins Spiel.



Da bis zur semantischen Analyse die Rechnereigenschaften nicht berücksichtigt wurden, kann man die Ergebnisse dieses Schrittes auf verschiedenartige Rechner übertragen (portieren).

Im Rahmen der Codeerzeugung – auch Synthese genannt – wird der Zwischencode, der bei der semantischen Analyse erzeugt wurde, in Objektcode, also in die Maschinensprache des jeweiligen Zielrechners übersetzt.



Dabei müssen die Eigenheiten des jeweiligen Zielbetriebssystems beispielsweise für die Speicherverwaltung berücksichtigt werden. Soll der erzeugte Objektcode auf einem anderen Rechnertyp als der Compiler laufen, so wird der Compiler als „Cross-Compiler“ bezeichnet.

### 5.1.6 Laufzeitsystem

Ein **Laufzeitsystem** (auf Englisch „**runtime system**“) enthält alle Programmfragmente, die zur Ausführung irgendeines Programms einer Programmiersprache notwendig sind, für die aber gar nicht oder nur sehr schwer direkter Code durch den Compiler erzeugt werden kann, oder für die direkt erzeugter Code sehr ineffizient wäre.



Dazu gehören alle Interaktionen mit dem Betriebssystem, wie beispielsweise Ansteuerung und Abfrage von Hardware-Ports, Abfangen von Signalen und das Einbinden von dynamischen **Bibliotheken** zur Laufzeit. Auch gehören dazu Speicherverwaltungsroutinen für den Heap (siehe Kapitel [→ 18](#)).

In der Programmiersprache C werden die Ein-/Ausgabe-Operationen normalerweise über Bibliotheken angesprochen. Wie diese Bibliotheken implementiert sind, ist je nach System unterschiedlich, weswegen die Ein-/Ausgabe-Operationen in C nicht zum eigentlichen Laufzeitsystem gezählt werden können.

Zum Laufzeitsystem gehören im Allgemeinen aber alle Prüfungen der dynamischen Semantik, kurz eine ganze Reihe von Fehler Routinen mit der entsprechenden Anwenderschnittstelle. Dies beinhaltet beispielsweise den sogenannten „Speicherabzug“. Auf Englisch wird dies „**core dump**“ genannt im Andenken an die magnetischen Kern- (Core-) Speicher, die zu Beginn der Datenverarbeitung genutzt wurden.

Auch besondere Sprachkonzepte wie Threads (parallele Prozesse) oder Exceptions (Ausnahmen) werden in aller Regel ebenfalls im Laufzeitsystem realisiert. Threads sind eine optionale Erweiterung des C11-Standards, siehe Kapitel [→ 23](#). Exceptions gibt es erst in C++.



## 5.2 Linker

Wenn ein Programm in der Programmiersprache C aus mehreren Modulen (Dateien) besteht, dann entstehen durch das Compilieren auch mehrere Dateien in Maschinensprache. Diese Dateien werden „**Objektdateien**“ genannt. Die Extension dieser Dateien ist `.obj` unter Windows und `.o` unter Unix.

Der Maschinencode dieser Objektdateien ist noch nicht selbständig ablauffähig. Zum einen, weil gewisse Bibliotheksfunktionen noch fehlen, zum anderen, weil die Adressen von externen Funktionen und Variablen aus anderen Objektdateien (Querbezüge) noch nicht bekannt sind. Der Linker bindet diese Objektdateien, die aufgerufenen Bibliotheksfunktionen und das Laufzeitsystem zu einer ablauffähigen Einheit zusammen.

Ein „**Linker**“ (zu Deutsch „**Binder**“) hat die Aufgabe, den nach dem Compilieren vorliegenden Objektcode in ein auf dem Prozessor ausführbares **Programm** (auf Englisch „executable program“, oder kurz „**executable**“) zu überführen.



Ist beispielsweise ein Programm getrennt in einzelnen Dateien geschrieben und in einzelne Objektdateien übersetzt worden, so werden die Objektdateien vom Linker zusammengeführt. Durch den Linker werden alle benötigten Teile zu einem ablauffähigen Programm gebunden. Hierzu gehört auch das Laufzeitsystem, das durch den jeweiligen Compiler eingebaut wird.

Variablen und Funktionen, welche im Quellcode geschrieben stehen, werden als „Speicherobjekte“ bezeichnet. Hier handelt es sich nicht um Objekte im Sinne der Objektorientierung, sondern um zusammenhängende Speicherbereiche. Beim Compilieren werden diese Speicherobjekte an relativen Adressen innerhalb der jeweiligen Objektdatei abgelegt. Wenn mehrere Objektdateien zu einer einzigen Datei zusammengefügt werden, werden diese Adressen dann vom Linker jeweils von einer – immer gleichen – Anfangsadresse ausgehend angeordnet.

Werden in der übersetzten Programmeinheit externe Variablen oder Funktionen, beispielsweise aus anderen Programmeinheiten oder aus Bibliotheken, verwendet, so kann der Übersetzer für diese Objekte noch keine endgültigen Adressen einsetzen. Vielmehr vermerkt der Compiler im Objektcode, dass an bestimmten Stellen Querbezüge vorhanden sind, an denen noch die Adressen der externen Objekte eingefügt werden müssen. Das ist dann die Aufgabe des Linkers.

Hier zeigt sich der große Vorteil von Bibliotheken: Die Übersetzung von Quellcode benötigt viel Zeit. Objekt-Dateien enthalten den Code einer Bibliothek in einer bereits übersetzten Form, so dass sie nicht erneut compiliert werden müssen. Der Linker kann so eine vorcompilierte Bibliothek genau gleich wie jede andere Objektdatei verarbeiten und muss einfach nur noch die Querbezüge auflösen.

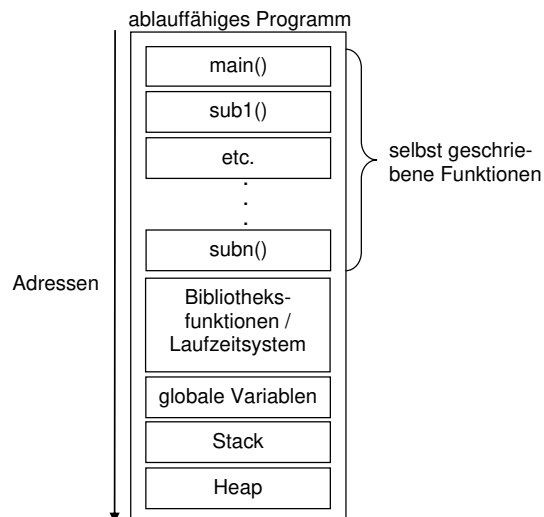
Ein Linker fügt die einzelnen Adressräume der Objektdateien, eingebundenen Standard-Bibliotheken und Funktionen des Laufzeitsystems so zusammen, dass sich die Adressen nicht überlappen, und löst die Querbezüge auf.



Hierzu stellt der Linker eine Tabelle her, welche alle Querbezüge (Adressen globaler Variablen, Einsprungadressen der Programmeinheiten) enthält. Damit können Referenzierungen von globalen Variablen oder von Funktionen aufgelöst werden. Durch den Linkvorgang wird ein einheitlicher Adressraum für das gesamte Programm hergestellt.

Das schlussendlich erstellte Programm hat unter Windows die Extension `.exe`, was für „executable program“ steht, auf Deutsch „ablauffähiges Programm“. Unter Unix gibt es keine Extension, jedoch wird auch dort das Programm als „executable“ bezeichnet. Eine ausführbare Datei wird unter Unix am sogenannten „executable-Flag“ erkannt.

Das folgende Bild zeigt als Beispiel den Adressraum eines ablauffähigen Programms:



## 5.3 Lader

Mit dem **Lader** (auf Englisch „**loader**“) wird das Programm in den Arbeitsspeicher des Computers geladen, wenn es gestartet wird.



Bei PCs mit modernem Speicherverwaltungssystem liegen die ablauffähigen Programme in Form eines verschiebbaren (auf englisch relocatable) Maschinencodes vor. Von einem verschiebbaren Maschinencode spricht man dann, wenn die Adressen des ablauffähigen Programms einfach von null an durchgezählt werden, ohne festzulegen, an welcher Stelle das Programm im Arbeitsspeicher liegen soll. Die Aufgabe des Laders besteht somit darin, den Code sowie die Daten an die richtigen Stellen im Arbeitsspeicher, die vom Betriebssystem bestimmt werden, abzulegen.

Der Lader lädt nicht nur den Code in den Arbeitsspeicher, sondern stellt auch sogenannte „Umgebungsvariablen“ dem Programm zur Verfügung und übernimmt Start-Argumente, welche an das Programm gesendet werden sollen. Mehr dazu kann in Kapitel [→ 17.1](#) nachgelesen werden.

Er setzt zudem das dynamische Laufzeitsystem auf, welches für die Speicherverwaltung, die Signalverarbeitung und das Laden dynamischer Bibliotheken zuständig ist.

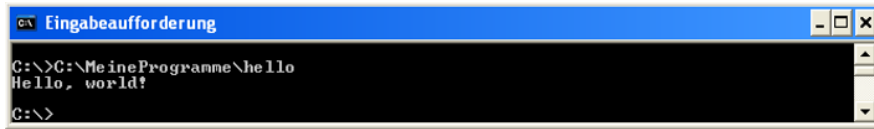
Wenn das Programm vollständig geladen ist, wird es vom Lader gestartet, indem die `main()`-Funktion aufgerufen wird.



Die ursprünglichste Art, ein lauffähiges Programm zu starten, ist mithilfe einer **Konsole**. Unter Windows muss hierfür beispielsweise die „Eingabeaufforderung“ gestartet werden und dort der Programmname – gegebenenfalls mit einer Pfadangabe – eingegeben werden (Die Extension `.exe` kann bei der Eingabe des Dateinamens auch weggelassen werden). Unter UNIX wird das gleiche Kommando in einer Shell abgesetzt.

```
C:\MeineProgramme\hello.exe
```

Das Programm läuft sodann in der Umgebung (auf Englisch „environment“) dieser Konsole. Das bedeutet, dass Ausgaben, die das Programm bei der Ausführung erzeugt, ebenfalls in der Konsole ausgegeben werden.



```
C:\>C:\MeineProgramme\hello
Hello, world!
C:\>
```

Das Laden, Starten und Ausführen eines Programms kann jedoch in verschiedenen Umgebungen erfolgen:

- Ganz bequem über ein Tippen mit dem Finger auf den Handybildschirm oder mittels Doppelklick mit einer Maus auf ein Programmsymbol.
- Über ein Script, welches beispielsweise ein Systemadministrator bereitstellt. Damit werden häufig wiederkehrende Ablaufpläne wie beispielsweise Installationen vorgenommen.
- Über eine integrierte Entwicklungsumgebung, häufig sogar unter der Kontrolle eines Debuggers.

In Umgebungen, welche kein hochentwickeltes Betriebssystem besitzen, kann das Laden eines Programms eine durchaus nicht triviale Angelegenheit sein, beispielsweise wenn ein Programm von einem Entwicklungs-PC erst auf ein Steuergerät geladen werden muss, um dort ausgeführt zu werden.

## 5.4 Integrierte Entwicklungsumgebungen und Debugger

Um die Programmentwicklung komfortabler zu gestalten, wird ein Programm heutzutage üblicherweise auf einer sogenannten „**Integrierten Entwicklungsumgebung**“ (auf Englisch „integrated development environment“, kurz **IDE**) entwickelt.



In einer integrierten Entwicklungsumgebung werden alle Aspekte der Programmierung in einem einzigen (integrierten) Tool verpackt. Dazu gehört normalerweise ein komfortabler Editor zur Eingabe der Programmtexte, Zugriff auf ein Quellcode-Versionierungs-System, ein Debugger und verschiedene Analyseprogramme, um beispielsweise den Speicherplatzverbrauch oder die Performance eines Programmes zu messen.

Um das Programm zu erstellen, beinhaltet eine Entwicklungsumgebung zudem mindestens einen Präprozessor, Compiler und Linker. Zusätzlich gibt es jedoch heutzutage auch die Möglichkeit, beliebige Skripte vor oder nach den einzelnen Stationen auszuführen. Dabei wird beispielsweise die gesamte Projektstruktur automatisch aufgesetzt, es werden Dateien kopiert, Datenbanken aufgesetzt, Zugriffsrechte von Dateiordnern verändert, grafische Benutzeroberflächen automatisch erstellt usw. Der gesamte Prozess der Programmerstellung wird als „**build**“ bezeichnet.

Darüber hinaus ist in einer Integrierten Entwicklungsumgebung oftmals eine Projektverwaltung integriert. Dabei kann ein Projekt aus mehreren Programmmodulen (Dateien) bestehen, welche getrennt entwickelt werden. Um das ausführbare Programm zu erzeugen, müssen alle Module durch den Compiler getrennt übersetzt und durch den Linker gebunden werden. Die Projektverwaltung sorgt dafür, dass bei Änderungen eines einzelnen Moduls nur dieses Modul übersetzt und mit den anderen, nicht geänderten Modulen neu gebunden wird.

Nach dem Laden wird das erstellte Programm normalerweise direkt gestartet. Formal korrekte Programme können jedoch logische Fehler enthalten, die sich erst zur Laufzeit und leider oftmals erst unter bestimmten Umständen während des Betriebs eines Programms herausstellen. Um diese Fehler analysieren und beheben zu können, möchte man den Ablauf des Programms während der Fehleranalyse exakt beobachten. Dazu dienen Hilfsprogramme, die als „Debugger“ bezeichnet werden.

Mit Hilfe eines **Debuggers** kann man Programme laden und gezielt starten, an beliebigen Stellen anhalten (sogenannte „Haltepunkte“ setzen), Programme nach dem Anhalten fortsetzen, Programme Schritt für Schritt ausführen sowie Speicherinhalte anzeigen und gegebenenfalls verändern.



Der Name „Debugger“ kommt vom Englischen „to debug“, was auf Deutsch soviel heißt wie „entwanzen“. Angeblich kommt dieser Begriff aus der Zeit der Lochstreifen, als nach einem fehlerhaften Programmlauf festgestellt wurde, dass beim Durchlaufen der Lochstreifen durch die Lesewalze ein Käfer genau an der Stelle eines Loches eingeklemmt und zerdrückt wurde, womit das Loch verdeckt wurde und das Lesegerät anstelle einer 1 eine 0 las. So entstand der umgängliche Name „Bug“. Heutzutage wird der Begriff „Debugging“ ganz allgemein benutzt im Sinne von Fehler suchen und entfernen.

Debugger ersetzen nicht den systematischen Test von Programmen zum Detektieren von Fehlern. Sie helfen jedoch dabei, zu aufgetretenen Fehlern die Ursache in einem Programm zu lokalisieren.