Make & CMake

Seminararbeit »Effiziente Programmierung «

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Veit-Hendrik Schlenker
E-Mail-Adresse: 4schlenk@informatik.uni-hamburg.de
Matrikelnummer: 6701608
Studiengang: Informatik

Betreuer: Christian Hovy

Hamburg, den 12.4.2017

Inhaltsverzeichnis

1 Einleitung

Computer sind seit jeher dazu da, uns Arbeit abzunehmen. Es braucht nur ein Programm, was das Problem löst. Doch je komplexer die Probleme werden, desto komplexer werden auch die Programme. Und damit steigt auch der Aufwand, die Einzelteile der Programme zu einem lauffähigen Endprodukt zusammenzuführen. Damit Programmierer sich auf die Aufgabe des Programmierens konzentrieren können, nehmen Buildsysteme ihnen diese Arbeit ab.

Diese Ausarbeitung zum Vortrag »Make & CMake« im Rahmen des Seminars »Effiziente Programmierung« erklärt den Sinn und Zweck von Buildsystemen und zeigt anhand von Make und CMake die Funktionsweise und das Prinzip, mit dem Fokus auf C und C++.

Zuerst wird ein Überblick darüber gegeben, wie Software überhaupt entsteht, und welche Probleme sich dabei ergeben und wie man diese mit Buildsystemen umgehen kann.

Mittels Make erfolgt eine erste Einführung in ein Buildsystem und welche Vorteile es gegenüber einfachen Shellscripten bietet. Die Syntax wird mithilfe eines Beispiels erklärt.

Aufbauend darauf wird CMake erläutert, welches anders als Make arbeitet und damit auch eigene Vor- und Nachteile mit sich bringt. Auch hier erfolgt auch eine Einführung an die Syntax anhand eines Beispiels.

Zum Schluss erfolgt eine Fazit der behandelten Themen.

2 Buildsystem

2.1 Wie wird Software gebaut?

Unter dem Oberthema »Software bauen« verstehen viele den Prozess, Quellcode zu schreiben um ein fertiges ausführbares Produkt zu erhalten. Was dabei häufig übersehen wird, sind die vielen Zwischenschritte die aus Quellcode ausführbare Binärdateien entstehen lassen. Meist wird auch fast alles, was zu diesem Schritt gehört in einem einzigen Schritt zusammengefasst, dem »Bauen« oder »Kompilieren« was viele IDE als einzelne Funktion anbieten.
Der Prozess des Kompilierens unterscheidet sich von Programmiersprache zu Programmiersprache. Damit ein allgemeiner Überblick gegeben werden kann, wird anhand von C++ beispielsweise gezeigt, welche Schritte erfolgen. Programme die nur als Quellcode vorliegen, müssen meist noch mit Befehlen wie `make` oder `./configure` erzeugt werden, wonach dann aber das gewünschte Programm zur Verfügung steht. Beide Befehle gehen entweder an Make oder CMake und sie dienen dazu, die einzelnen Schritte bis zur fertigen Software auszuführen.
Da es wichtig ist zu verstehen, welche Schritte beim Bauen von Software durchgeführt werden, um diese durch ein Buildsystem zu ersetzten, werden die einzelnen Schritte hier nochmal erklärt.

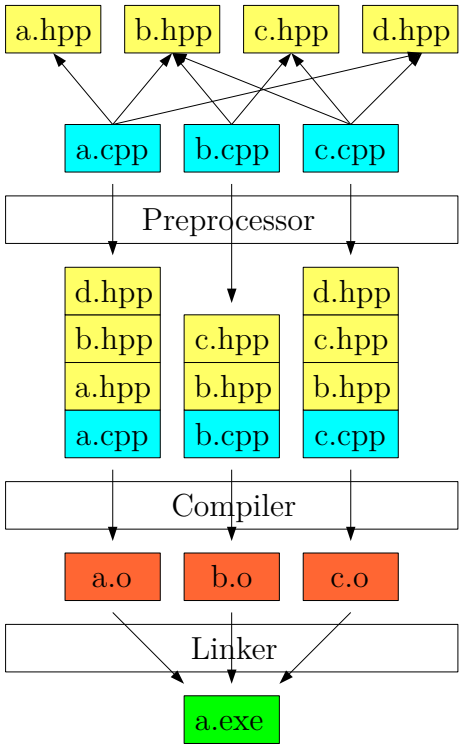

Abbildung 2.1: Software Buildprozess

1. Quellcode schreiben
   Die *.cpp Dateien und ihre dazugehörigen *.hpp Dateien, die die entsprechenden Funktionsdeklarationen enthalten, erstellen.

2. Präprozessor
   Löst Befehle auf und inkludiert Funktionsdeklarationen entsprechend der Headerfiles.

3. Compiler
   Kompiliert die Quelldateien in ein für den Computer optimiertes und ausführbares Format.

4. Linker
   Fügt die vom Compiler erzeugten Objekte zu einem ausführbaren Programm zusammen

Der **Präprozessor** löst Befehle auf wie `include`, `#define`, `#if` oder `#ifdef` durch einfache Textersetzung. Das `include` wird ausgeführt, indem das entsprechende Headerfile in die Datei rein kopiert wird. `#define`, `#if` und `#ifdef` kann man unter dem Oberbegriff des »Makros« zusammenfassen. Sie können dazu dienen, häufig auftauchende Konstanten zu ersetzen:

```
1 #define BUFFER_SIZE 1024
2 foo = (char *) malloc (BUFFER_SIZE);
```

Oder um Debug-Nachrichten bei gesetzten Flags zu ermöglichen:

```
1 #if (DEBUGLEVEL >= 1)
2 #  define print1 printf
3 #else
4 #  define print1(...) (0)
5 #endif
```

Der **Compiler** übersetzt den vorbereiteten Code in ausführbaren Binärcode. Aus einem

```
1 int i = 2;
```

in C wird

```
1 movl    $2, -4(%ebp)
```

in Assembler und daraus der Code für die CPU entsprechend der ISA. Es finden auch weitere Zwischenschritte statt wie die Optimierung des Codes, wo es möglich ist. Dies kann zum Beispiel das Entfalten einer Schleife sein, die nur zweimal durchlaufen wird.

Der **Linker** ist der letzte Schritt und dient dazu, all die externen Aufrufe an Bibliotheken einzubinden. Dazu gibt es zwei Möglichkeiten. Das statische Linken, was alle Bibliotheken fest mit dem Programm verbindet, wodurch sie zwar immer verfügbar sind, aber auch mehr Speicherplatz benötigen. Im Falle eines Updates einer Bibliothek ist auch ein neukompilieren des Programmes erforderlich.
Weit häufiger verbreitet ist daher heute das dynamische Linken, bei dem das Betriebssystem die nötigen Bibliotheken nachlädt.
Nach diesen Schritten liegt ein ausführbares Programm vor.

2.2 Warum sind Buildsysteme sinnvoll?

Ein Buildsystem nimmt nun idealerweise die Schritte 2-4 ab, indem es sie zu einem Befehl zusammenfasst weshalb diese Schritte nicht bei jedem Build erneut ausgeführt werden müssen. Der große Vorteil dabei ist, dass damit auch ein wiederholtes Abtippen von gleichen Befehlen wegfällt, denn dieses wiederholte eintippen ist nicht nur langweilig, sondern kann auch zu Problemen führen. Es passiert bei derartig monotonen Aufgaben häufig, dass sich kleinere Fehler einschleichen. Wenn unwissentlich ein Befehl falsch eingetippt wird, gibt es meistens auch die passende Warnung dazu. Der falsche Befehl

könnte aber auch auf einmal einem anderen entsprechen wodurch ein anderer Befehl ausgeführt wird, der wiederum ein anderes Produkt erzeugt mit einer etwas anderen Verhaltensweise. Dadurch kann es passieren, dass viel Zeit damit verloren geht, Fehler zu finden, die gar nicht da sind. Daher kann ein Buildsystem nicht nur dazu beitragen, Zeit zu sparen, sondern es kann sich auch darauf verlassen werden, dass das Endprodukt sich nur durch die Änderungen im Quellcode von der vorherigen Version unterscheidet.

3 Make

3.1 Überblick

Die Vorgänger von den heutigen Buildsystem wurden schon recht früh verwendet, da es bei größeren Produkten wie Betriebssystemen und Kerneln nicht mehr zumutbar war, das jeder Entwickler sich alle Befehle und Pfade merken sollte. Daher wurden diese in Shellscripte übernommen, die automatisiert die nötigen Befehle ausführten.
Wie so häufig in der Softwareentwicklung bleibt es aber nicht bei der einen Anforderung. Denn häufig hatte man nicht nur ein »Buildtarget« sondern mehrere wie Release, Staging und Debug, die alle unterschiedliche Compiler-Flags verwendeten und daher auch unterschiedlich gebaut werden mussten.
Neben den bisher erwähnten Vorteilen von Buildsystemen, die hauptsächlich dem Komfort dienen, bietet Make aber auch Vorteile die sich von einem simplen Shell-Script unterscheiden. Make ist in der Lage, Änderungen über den Änderungszeitpunkt der Datei zu erkennen und diese neuen Dateien -und nur die- neu zu erzeugen und auch alles, was auf dieser Datei aufbaut. Diese Fähigkeit war auch einer der Katalysatoren für die Entwicklung von Make selber:

> Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.
> – Stuart Feldmann [Ray03]

Make erschien am 24.4.1976 das erste Mal als Teil von Unix und wurde von Stuart Feldmann entwickelt [mak]. Seit dem hat es viele Abkömmlinge bekommen wie dmake für OpenOffice, GNU Make was das ursprüngliche Make um conditionals, foreach erweitert und im Linux Kernel eingesetzt wird, und die Visual Studio Implementation nmake.

3.2 Funktionsweise

Vereinfacht kann man Make damit zusammenfassen, das A nach B konvertiert wird. Das Prinzip lässt sich auf alles Mögliche anwenden, automatisches konvertieren, filtern und skalieren eines Bildes beispielsweise. Nur dass hier eben Quellcode zu ausführbarer Software wird.
Um genauer auf die Funktionsweise eingehen zu können, wird rückwärts vom fertigen Programm aus gedacht. Damit das Programm entstehen kann, müssen Bedingungen erfüllt sein. Diese beinhalten den kompilierten Quellcode und die Standardlibrary, um sie zum fertigen Produkt zu linken. Damit der kompilierten Quellcode vorliegen kann, müssen die einzelnen Dateien mit den passenden Headern kompiliert werden. Meistens hängen mehrere Dateien zusammen, also Objekt B kann erst kompiliert werden, wenn A schon da ist. Diese Abhängigkeiten kann Make auflösen und die Dateien in der nötigen Reihenfolge kompilieren. Bei weiteren Durchläufen muss dann nur noch das, was sich geändert hat, neu kompiliert werden.

3.3 Syntax & Beispiele

Wenn Make gestartet wird, sucht es im aktuellen Ordner zuerst nach Makefiles, das sind einfach Textdateien mit Namen wie GNUMakefile, makefile oder Makefile. Diese werden geladen und doe Kommandos darin ausgeführt. Es kann zwischen drei großen Arten von Kommandos unterscheiden:

1. Targets (Ziele)
2. Prerequisites (Vorbedingungen)
3. Commands (Kommandos)

Targets sind die Ziele die durch einen Kommandoblock erzeugt werden. Wird in dem Block eine Datei `hello.o` erzeugt, so ist das Ziel ebenfalls `hello.o`.
Es gibt auch Ziele, die keine Dateien erzeugen. Sehr häufig ist dabei das `clean`-Kommando was die erzeugten Dateien löschen soll. **Vorbedingungen** sind die Ziele, die erzeugt werden müssen, bevor das aktuelle Ziel erzeugt werden kann. Zum Schluss kommen die eigentlichen **Kommandos** die zum Erzeugen des Zieles ausgeführt werden sollen. Allgemein sieht die Syntax so aus:

```
1 target target target : prerequisite prerequisite
2 (tab) command
3 (tab) command
```

Listing 3.1: Make Syntax

Zu beachten ist das `tab` vor den Befehlen. Es ist nicht ersetzbar mit Leerzeichen oder kann ganz weggelassen werden, da der Makesyntax verlangt, dass vor den Kommandos ein Tab-Charakter steht.

Um die Syntax besser verstehen zu können, wird sie an Hand eines konkreten Beispieles erläutert:

```
1  edit : main.o kbd.o
2      cc -o edit main.o kbd.o
3
4  main.o : main.c defs.h
5      cc -c main.c
6
7  kbd.o : kbd.c defs.h command.h
8      cc -c kbd.c
9
10 .PHONY: clean
11 clean :
12     rm edit main.o kbd.o
```

Listing 3.2: Make Beispiel

In diesem Beispiel gibt es vier Ziele: `edit, main.o, kbd.o` und `clean`. Wenn der Aufruf von `make` parameterlos erfolgt, wird das erste Ziel kompiliert. Das ist hier `edit`. `edit` hat aber zwei Vorbedingungen: `main.o` und `kbd.o` die zuerst kompiliert werden müssen. Daher sucht Make nach diesen Zielen, und prüft ihre Vorbedingungen. Diese sind nur Dateien und da sie nicht als Ziele definiert sind, muss Make nur prüfen ob diese Dateien vorhanden sind und kann dann die entsprechend Kommandos ausführen. Nachdem das passiert ist, kann auch `edit` erzeugt werden.
Sollte man nun an der Datei `main.c` was ändern, erkennt Make das bei erneutem Aufruf und kompiliert `main.o` und `edit` neu, aber `kbd.o` nicht. Wird Make aufgerufen, ohne die Dateien geändert zu haben, wird auch nichts passieren.
Damit man nun alle erzeugten Dateien auch einfach wieder löschen kann, wurde hier `clean` als Ziel eingeführt, das sich über `make clean` aufrufen lässt. Da `clean` gerade keine Dateien erzeugt, wurde mit `.PHONY: clean` bestimmt, dass `clean` immer bei Aufruf ausgeführt wird, auch wenn eine Datei namens `clean` in dem Ordner existieren sollte.
Ein sehr häufig benutztes Element in Makefiles sind Variablen, um beispielsweise immer wieder benötigte Standardelemente nicht erneut schreiben zu müssen. Variablen in Makefiles funktionieren ähnlich wie `#define` im C-Präprozessor, da sie einfach nur eine Textersetzung darstellen. Damit sind auch solche Konstrukte problemlos möglich:

```
1 foo = c
2 prog.o : prog.$(foo)
3 $(foo)$(foo) -$(foo) prog.$(foo)
```

was zu

```
1 prog.o : prog.c
2 cc -c prog.c
```

übersetzt wird.

4 CMake

4.1 Überblick

Wie zuvor erwähnt, hat Make zwar sehr viele Vorteile, aber mindestens einen Nachteil: Es ist nicht plattformunabhängig. Es ist zwar mit jedem Unix System kompatibel, aber nicht mit Windows. Und ebenso kann Make nicht mit Projektmappen umgehen, wie sie viele IDEs wie Visual Studio oder Xcode verwenden.
Eine der frühen Lösungen diese Problems war das VTK Buildsystem. Dies entstand um 1999 [cmaa] und enthielt neben einem kleinen Konfigurationsscript auch eine ausführbare Datei namens `pcmaker`, welches Unix Makefiles in Windows NMake Files wandelte. Das Problem war aber, dass nicht nur der `pcmaker` bei Aktualisierungen immer wieder neu als ausführbare Datei ins Versionsverwaltungssystem hochgeladen werden musste, sondern auch, dass die Entwicklung in Windows häufig in IDEs stattfand, der PCMaker aber ein Kommandozeilenprogramm war. Dies konnte dazu führen, dass es für die Benutzer einfacher war, die Projektdateien zu ändern, als erneut den pcmaker zu starten.
Um diese Probleme eleganter zu lösen und dabei so vielseitig wie Make zu bleiben, entstand Mitte 2000 CMake. Im Vergleich zu anderen Systemen der Zeit, hat CMake einige Besonderheiten. Es gibt keine weitere Abhängigkeit als den C++ Compiler selber, womit CMake mit jeder Plattform die auch C++ unterstützt, kompatibel ist. Mittlerweile kann CMake populäre Bibliotheken wie `bzip` selbständig finden und sie mit dem Ziel verlinken [CMab].
Der wichtigste Unterschied ist aber, dass CMake selber gar nicht die Software baut. CMake könnte eher als »Metabuildsystem« verstanden werden, welches die Makefiles oder Projektmappen baut, die andere Programme dann zum Bauen der Software verwenden. Prinzipiell kann man sich das so vorstellen, dass CMake, ähnlich wie viele Compiler heutzutage, die Bauanweisungen in eine Zwischensprache übersetzt und optimiert, um dann daraus, mittels verschiedener Generatoren, das Endprodukt zu bauen. Das kann ein Makefile, eine VisualStudio oder XCode Projektmappe oder auch ein Eclipse Projekt sein. Der Vorteil dieser Herangehensweise ist, dass eine Änderung im Buildprozess schnell auf andere Plattformen übertragen ist.

4.2 Funktionsweise

Es gibt zwei wichtige Zwischenschritte bei CMake, den `Configure` und den `Generate` Schritt. Beim `Configure` wird eine eventuell existierende `CMakeCache.txt` aus vorherigen Durchläufen eingelesen, dann wird ausgeführt, was in der CMakeLists.txt steht wodurch eine interne Repäsentation des Projekes entsteht. Zum Schluss wird die `CMakeCache.txt` geschrieben, die verschiedene Umgebungsvariablen wie Konfigurationen und Bibliothekspfade enthält. Der `Generate` Schritt wandelt die vorher erstellte interne Repräsentation des Projektes mittels Generatoren in Makefiles oder Projektmappen. Diese Ausgabe kann dann dazu verwendet werden, das eigentlich Programm zu kompilieren.

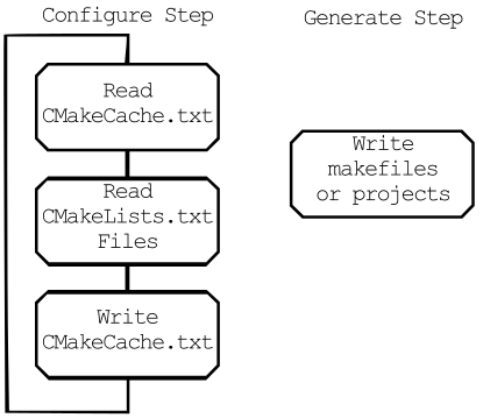

Abbildung 4.1: CMake Schritte [Mar]

4.3 Syntax & Beispiele

CMake hat zwar einige Ähnlichkeiten zu Make, aber viel mehr Unterschiede. In dem Beispiel wird ein simples Projekt betrachtet, welches eine c-Datei im Hauptverzeichnis hat.

```
1 cmake_minimum_required (VERSION 2.6)
2 project (HelloWorld)
3 add_executable(HelloWorld HelloWorld.c)
```

Listing 4.1: CMake Beispiel

Die ersten zwei Zeilen bestimmen zunächst das Minimum der benötigten CMake-Version mittels `cmake_minimum_required()` und einen Projektnamen durch `project()`. Hier fällt schon ein großer Unterschied in der Skriptsprache von CMake auf. Während bei Make auf die Ziele und Vorbedingungen die Terminalbefehle folgen, sind bei CMake alle Befehle in dem Format `funktionsname(variablen)` geschrieben.
Durch `add_executable` passiert das Wichtigste, es wird ein Ziel namens `HelloWorld` aus der Datei `HelloWorld.c` generiert. Das praktische dabei ist, dass CMake keine spezielle Angabe braucht, welcher Compiler verwendet werden soll. Dies wird aus der Dateiendung selber ausgelesen.

Das Beispiel wird nun fortgeführt, indem eine Abhängigkeit durch eine Bibliothek in einem Unterordner `MathFunctions` hinzufügt wird.

```
1 cmake_minimum_required (VERSION 2.6)
2 project (HelloWorld)
3 include_directories("${PROJECT_SOURCE_DIR}/MathFunctions")
4 add_subdirectory(MathFunctions)
5 add_executable(HelloWorld HelloWorld.c)
6 target_link_libraries (HelloWorld MathFunctions)
```

Listing 4.2: CMake Beispiel

Dieser Unterordner enthält neben der Quelldatei `mysqrt.cpp` auch selber nochmals eine CMakeLists.txt

```
1 add_library(MathFunctions mysqrt.cpp)
```

Listing 4.3: MathFunctions

Es sind drei Funktionsaufrufe dazugekommen:
`include_directories` fügt einen Order den Suchpfaden nach Header-Dateien hinzu.
`add_subdirectory` erlaubt es in anderen Verzeichnissen als in dem wo auch die CMakeLists.txt liegt nach CMake Dateien zu suchen, bei uns das Unterverzeichnis `MathFunctions`.
`target_link_libraries` verlinkt eine ausführbare Datei `HelloWorld` mit der Bibliothek `MathFunctions`.
Im Vergleich mit Make fällt auf, dass CMake nicht die Kommandozeilenbefehle zu den passenden Zielen hält wie Make, sondern dass Ziele über die Funktion `add_executable` hinzugefügt werden, und der richtige Compiler gewählt wird

5 Zusammenfassung

Moderne Buildsysteme haben einen langen Weg hinter sich, bis sie zu dem wurden, was sie heute sind. Mittlerweile ist es üblich und sinnvoll, sie einzusetzen und manchmal auch unabdingbar. Sie ermöglichen es schneller, fehlerfreier und einfacher Software zu bauen. Denn dank Buildsystemen ist es heutzutage möglich, auch derartig komplexe Softwaregebilde wie den Linux-Kernel, der über rekursiv aufgerufene makefiles erzeugt wird, zu bauen, ohne komplizierte und lange Befehlsketten eingeben zu müssen.
Neben Make und CMake existieren auch Alternativen, wie zum Beispiel Gradle, mit dem Android Apps heutzutage standardmäßig gebaut werden oder Ninja, worauf das Android Open Source Project, also das was die Basis von Android darstellt, umgestellt wird [AOS]. Selbstverständlich gibt es einen Generator für CMake, der aus den CMakeLists.txt auch Ninja Makefiles erzeugen kann.

Die Dateien zu den Beispielen sind unter [Git] verfügbar.

Literaturverzeichnis

[AOS] *AOSP with Ninja.* https://groups.google.com/forum/#!topic/android-platform/Hhl_4hfOONg, Abruf: 25.02.2017. – (besucht am 28.02.2017)

[cmaa] *About CMake.* https://cmake.org/overview/, Abruf: 26.02.2017

[CMab] *CMake How To Find Libraries.* https://cmake.org/Wiki/CMake:How_To_Find_Libraries, Abruf: 28.02.2017

[Git] *Beispieldateien.* https://gitlab.com/vhschlenker/semvortrag-dateien

[mak] *Unix 7th Edition.* http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/make/ident.c, Abruf: 22.02.2017

[Mar] Martin, B. Hoffman , K.: *CMake, In: The Architecture of Open Source Applications.* http://www.aosabook.org/en/cmake.html, Abruf: 26.02.2017

[Ray03] Raymond, Eric S.: *The Art of Unix Programming.* PRENTICE HALL COMPUTER, 2003 http://www.ebook.de/de/product/3259137/eric_s_raymond_the_art_of_unix_programming.html. – ISBN 0131429019