Automatisches Übersetzen und Binden *(Ubuntu Linux 20.04, g++-9.4)*

Mit dem Systemkommando **make** ist es u.a. möglich bei Programmänderungen automatisch nur die geänderten Dateien zu übersetzen und zu einer ausführbaren Objektcodedatei zu binden. Die dazu benötigten Abhängigkeiten der Dateien voneinander und die Kommandos werden aus einer vom Benutzer zu erstellenden Datei (Makefile) gelesen. Der Zeitpunkt der letzten Änderung der einzelnen Dateien bestimmt dann, welche Kommandos tatsächlich ausgeführt werden.

make [–**f** *makefile*] [–**n**] [–**s**] [*target*] [*macro=value ...*]

Führt die *target* entsprechenden Kommandos in der Datei *makefile* (Voreinst.: `makefile`) aus. Im Falle der Option –**n** werden die Kommandos nur angezeigt, nicht aber ausgeführt, während für –**s** die Kommandos nur ausgeführt, nicht aber angezeigt werden. *target* ist oft eine ausführbare Objektcodedatei und die Kommandos in *makefile* geeignete Übersetzungs- und Bindekommandos.

In einer vereinfachten Darstellung ist *makefile* aus folgenden Elementen aufgebaut.

target: *file(s)*	Zieldatei und Dateien, deren Änderung eine Änderung der Zieldatei notwendig macht
Tab *Kommando*	auszuführendes Kommando, falls eine Datei, von denen die Zieldatei abhängt, *nach* der letzten Änderung der Zieldatei geändert wurde

Anweisungen können in die nächste Zeile fortgesetzt werden, wenn die Zeile mit `\` beendet wird.
Fortgesetzte Kommandozeilen müssen zusätzlich mit Tab beginnen.
Kommentare können durch Voranstellen von `#` in die Datei eingefügt werden.
Die Ausführung der Kommandos erfolgt rekursiv entsprechend den Abhängigkeiten.
Fehlen Angaben in *makefile*, so werden Abhängigkeiten und Kommandos nach voreingestellten Regeln benutzt.

Bsp. für eine Datei `makefile`

```
# Erzeugung einer ausfuehrbaren Zieldatei prog aus main.cpp, ld.cpp und ld.h
#
prog: main.o ld.o
Tab c++ -g main.o ld.o -o prog    # Binden der Objektcodedateien
main.o: main.cpp ld.h
Tab c++ -g -c main.cpp            # Objektcodedatei main.o erzeugen
ld.o: ld.cpp ld.h
Tab c++ -g -c ld.cpp              # Objektcodedatei log2.o erzeugen
clean:
Tab rm -f main.o ld.o prog        # Aufraeumen
```

Ausführen der Befehle: `make prog` (oder `make`)

Makrodefinition

name = *string* definiert einen Namen für eine Zeichenkette, der innerhalb des Makefiles für `$(`*name*`)` eingesetzt wird.
Beim `make`-Aufruf angegebene Makrodefinitionen haben Vorrang vor Definitionen im Makefile oder in impliziten Regeln.

Vordefinierte Namen sind u.a.

`$@`	Name des aktuellen Ziels
`$*`	Name des aktuellen Ziels ohne Dateierweiterung (in impliziten Regeln)
`$<`	Name der abhängigen Datei (in impliziten Regeln)

Implizite Regeln

Implizite Regeln werden durch ein Ziel der Form *.ext1*[*.ext2*] definiert, denen Kommandos folgen, die zu einer Datei mit der Endung *.ext1* eine Datei mit der Endung *.ext2* erzeugen. Z.B.:

```
.cpp.o:
[Tab]c++ $(CXXFLAGS) -c $<
```

Implizite Regeln werden nur angewendet, wenn für ein Ziel keine explizite Regel spezifiziert ist (und die Endungen in der `.SUFFIXES`-Liste aufgeführt sind). Sie lassen sich mit `make -p` ausgeben.

Beispiel für eine Datei `makefile` *:*

```
OBJ = main.o ld.o
CXXFLAGS = -g
#CXXFLAGS = -O
#
prog: $(OBJ)
[Tab]cc $(CXXFLAGS) $(OBJ)-o $@
clean:
[Tab]rm -f $(OBJ) prog
#
main.o: main.cpp ld.h
[Tab]c++ $(CXXFLAGS) -c main.cpp
ld.o: ld.cpp ld.h
[Tab]c++ $(CXXFLAGS) -c ld.cpp
```

Wegen der angenommenen impliziten Regel können die beiden letzten cc-Aufrufe entfallen.
Mit `make prog CXXFLAGS=-O` oder `make 'CXXFLAGS=-O -g'` läßt sich ohne Änderung des Makefiles das Programm optimiert und ohne bzw. mit Debug-Informationen übersetzen.

Der C/C++-Präprozessor

Arbeitsphasen

1. Ersetzen der Dreizeichenfolgen

`??(` → `[`	`??)` → `]`	`??=` → `#`
`??<` → `{`	`??>` → `}`	`??/` → `\`
`??'` → `^`	`??-` → `~`	`??!` → `|`

2. Verbinden aufeinanderfolgender Zeilen, falls die vorhergehende mit `\` endet.
3. Zerlegen in Präprozessor-Eingabesymbole, Ersetzen von Kommentaren durch Leerzeichen, Ausführen der Präprozessoranweisungen (insb. Makroexpansion).
4. Auswerten der Ersatzdarstellungen in Zeichen(ketten)konstanten, Verbinden benachbarter Zeichenkettenkonstanten.
5. Zerlegen in C/C++-Eingabesymbole

Präprozessoranweisungen (Auszug)

Alle Präprozessoranweisungen beginnen mit `#` am Zeilenanfang und enden am Zeilenende. (Zwischenraum vor `#` ist ebenfalls zulässig.)

`#include` <*file*>
`#include` "*file*"

> Die angegebene Header-Datei wird an der Stelle dieser Anweisung eingelesen. Sie wird an Stellen gesucht, die von der Implementierung abhängen (z.B. in Standardverzeichnissen). Die zweite Form dient zum Einlesen von Quellcodedateien. Nach ihr wird zuerst implementationsabhängig gesucht (z.B. in dem Verzeichnis, das die Datei mit `#include` enthält) und danach wie bei der ersten Form.

`#define` *name* [$token_1$ $token_2$ ...]
`#define` *name*($name_1$,$name_2$,...) [$token_1$ $token_2$ ...]

> Ersetzt im folgenden Text der Übersetzungseinheit den Makronamen *name* bzw. *name*(arg_1,arg_2,...) durch die angegebene Symbolfolge. Davor werden die eingesetzten Argumente ausgewertet, allerdings nicht, wenn dem Parameter ein `#` vorausgeht oder wenn zwischen zwei Parametern `##` steht. Danach werden in der so erzeugten Symbolfolge wieder Textersetzungen vorgenommen usw.
>
> Innerhalb von "..." bzw. ' ' findet kein Textersatz statt.
> `#`$name_i$ in der Symbolfolge wird ohne Auswertung durch "arg_i" ersetzt.
> `##` in der Symbolfolge verkettet nach Einsetzung der Argumente die unmittelbar benachbarten Symbole (ohne vorherige Auswertung).
> Ein weiteres `#define` für einen Makronamen ohne vorheriges `#undef` ist unzulässig, außer wenn die Definition identisch ist (bis auf Zwischenraum).

`#undef` *name*

> Löscht die Definition des Namens.

```
#if expression
⋮
[#elif expression
⋮           ]
[#else
⋮           ]
#endif
```

Der Text in dem `if`– bzw. `elif`–Zweig, für den *expression* zum ersten Mal $\neq 0$ ist, wird bearbeitet; der Text der anderen Verzweigungen wird ignoriert. Falls der Wert des Ausdrucks in allen `if`– und `elif`-Anweisungen gleich 0 ist, wird der Text des `else`–Zweigs bearbeitet und der Text der anderen Verzweigungen übergangen. Im Konstantenausdruck *expression* werden neben den in C/C++ üblichen Umwandlungen die Umwandlungen `int` $\rightarrow$ `long` und `unsigned int` $\rightarrow$ `unsigned long` vorgenommen.
Die Konstantenausdrücke `defined` *name* oder `defined (`*name*`)` werden durch 1 ersetzt, falls eine Makrodefinition für *name* existiert; andernfalls durch 0.

`#ifdef` *name*
`#ifndef` *name*

Äquivalent zu `#if defined` *name* bzw. zu `#if !defined` *name*.

Beispiele zur Präprozessorbenutzung

1)
```
cout << "Weiter?\?(j/n) ");
```

2)
```
char text[] = "ziemlich kurzer "
              "Text mit \n"
              "Zeilenumbruch"
```

3)
```
#define sqr(x) ((x)*(x))
int main()
{  :
  d = c/sqr(a+b)
   :
}
```

4)
```
#define max(a,b) ((a) > (b) ? (a) : (b)) // haeufig in C-Code
```

5)
```
#define xstr(s) str(s)
#define str(s) #s
#define TEXT Das ist ein Text

xstr(TEXT)
```

6)
```
#ifndef N
#define N 100
#endif
```

Oft im Zusammenhang mit einer Übersetzeroption, z.B. `c++ -DN=500 prog.cpp`

Zu den wichtigsten vordefinierten Makros gehören `__cplusplus` mit dem `long`-Wert `199711` für C++98 und `__STDC__` mit dem Wert 1 für ANSI-C.
Außerdem sind in der Regel Makros vorhanden, die dem Betriebssystem und dem Prozessor entsprechen. (Z.B. g++-9.4 unter Ubuntu Linux 20.04 (amd64): `unix`, `linux`, `__x86_64`).

7)
```
#if defined(sun) && !defined(BSD)
#include "solaris.h"
#elif defined(__hpux)
#include "hpux.h"
#else
#include "generic.h"
#endif
```