make / Makefiles (1)

Das Programm "make" ermöglicht es, die Schritte zu automatisieren, die nötig sind um ein Programm neu zu übersetzen. Dazu muss in einer Datei - dem Makefile - festgehalten werden, welche Schritte dazu nötig sind. Beispiel:

```
all: prog

prog: prog1.cc prog2.cc prog3.cc
	g++ -c -Wall -O2 -o prog1.o prog1.cc
	g++ -c -Wall -O2 -o prog2.o prog2.cc
	g++ -c -Wall -O2 -o prog3.o prog3.cc
	g++ -o prog prog1.o prog2.o prog3.o
```

Das Übersetzen erfolgt mit

```
# make
```

womit automatisch das erste Target, "all", ausgeführt wird. Es ist auch möglich make ein oder mehrere Targets zu übergeben, beispielsweise:

```
# make prog
```

Obiges Makefile ist noch nicht optimal, da immer alle Dateien neu übersetzt werden, wenn sich prog1.cc, prog2.cc oder prog3.cc ändern. Besser wäre:

```
all: prog

prog: prog1.o prog2.o prog3.o
	g++ -o prog prog1.o prog2.o prog3.o

prog1.o: prog1.cc
	g++ -c -Wall -O2 -o prog1.o prog1.cc

prog2.o: prog2.cc
	g++ -c -Wall -O2 -o prog2.o prog2.cc

prog3.o: prog3.cc
	g++ -c -Wall -O2 -o prog3.o prog3.cc
```

Hier kann "make" die Abhängigkeiten verwenden, um nur genau das zu übersetzen, was auch geändert wurde. Dies geht über das Änderungsdatum der Dateien.

make / Makefiles (2)

Der generelle Aufbau einer Regel ist:

```
<Target>: <Abhängigkeit1> ... <AbhängigkeitN>
	<Shellbefehl1>
	<Shellbefehl2>
	...
	<ShellbefehlM>
\ TAB! /
```

Beim Ausführen der Regel stellt "make" zunächst sicher, dass die Abhängigheiten auf dem neusten Stand sind, und wenn mindestens eine neuer als "<Target>" ist, wird dieses durch Ausführen der Shellbefehle neu erstellt.

Es gibt eine Datenbank von Regeln, die auf Dateiendungen basieren, und die dann greifen, wenn man "make" nicht sagt, wie etwas neu zu übersetzen ist. Damit kann man kurz schreiben:

```
CXXFLAGS = -Wall -O2

all: prog

prog: prog1.o prog2.o prog3.o
	g++ -o prog prog1.o prog2.o prog3.o
```

Oft ist es dennoch nötig die Abhängigkeiten anzugeben, auch wenn die eigentlichen Befehle aus der "make"-Datenbank kommen, beispielsweise, wenn ein gemeinsamer Header verwendet wird:

```
CXXFLAGS = -Wall -O2

all: prog

prog: prog1.o prog2.o prog3.o
	g++ -o prog prog1.o prog2.o prog3.o

prog1.o: prog1.cc prog.h

prog2.o: prog2.cc prog.h

prog3.o: prog3.cc prog.h
```

```
==============================================================================
make / Makefiles (3)
==============================================================================
Variablen (wie CXXFLAGS) können verwendet werden um mehrfach vorkommende oder
sich häufig ändernde Teile zu ersetzen.


CXXFLAGS = -Wall -O2
OBJS = prog1.o prog2.o prog3.o

all: prog

prog: $(OBJS)
        g++ -o prog $(OBJS)

prog1.o: prog1.cc prog.h

prog2.o: prog2.cc prog.h

prog3.o: prog3.cc prog.h

clean:
        rm -f prog $(OBJS)

================================================================================
Strukturen
================================================================================
Strukuren sind benutzerdefinierte Datentypen, die aus mehreren Feldern bestehen.

#include <stdio.h>

struct Bruch
{
    int zaehler;
    int nenner;
};

Bruch addiere (Bruch a, Bruch b)
{
    Bruch summe;
    summe.nenner = a.nenner * b.nenner;
    summe.zaehler = a.zaehler * b.nenner + a.nenner * b.zaehler;
    // kuerzen noetig!
    return summe;
}

int main()
{
    Bruch x = { 1, 2 }, y = { 1, 3 };
    Bruch z = addiere (x, y);

    printf ("1/2 + 1/3 = %d/%d\n", z.zaehler, z.nenner);
}
```



```
================================================================================
Gleitkommazahlen
================================================================================
Darstellung einer reellen Zahl als:

    f = s * m * 2^e

s   Sign (Vorzeichen)
m   Mantisse
e   Exponent


Bits         Sign        Mantisse     Exponent      gesamt
--------------------------------------------------------------
float         1             23            8             32
double        1             52           11             64
long double   1             64           15             80

================================================================================
typische Funktionen und Konstanten
================================================================================
#include <math.h>

sin (x), cos (x), tan (x)             trigonometrische Funktionen
asin (x), acos (x), atan (x)          arc sin (x), ...
sqrt (x)                              Wurzel x
pow (x, y)                            x ^ y
exp (x)                               e ^ x
log (x)                               natürlicher Logarithmus
fabs (x)                              Betrag x

M_PI                                  pi

================================================================================
Beispiel: wurzel2.cc
================================================================================

#include <stdio.h>
#include <math.h>

double wurzel2()
{
    double untere_grenze = 1;
    double obere_grenze = 2;

    while ((obere_grenze - untere_grenze) > 1e-15)
    {
        double mitte = (untere_grenze + obere_grenze) / 2;
        if (mitte * mitte < 2)
        {
            untere_grenze = mitte;
        }
        else
        {
            obere_grenze = mitte;
        }
    }
    return (untere_grenze + obere_grenze) / 2;
}
```

```
int main()
{
    printf ("Wurzel 2 = %.17g\n", wurzel2());
    printf ("sqrt(2)  = %.17g\n", sqrt (2));
}

================================================================================
Automatische und explizite Typkonvertierung
================================================================================
Bei Rechnungen mit Operanden unterschiedlichen Typs wird der Operand mit
geringerer Präzision in den Typ grösserer Präzision konvertiert, und die
Rechnung dann ausgeführt.

  char -> short -> int -> long long -> float -> double

    double d = 1 + 1 / 3;                   // => d = 1
    double d = 1 + 0.33;                    // => d = 1.33
    double d = 1 + 1 / 3.0;                 // => d = 1.33333...

Bei Zuweisungen eines Typs grösserer Präzision wird konvertiert, es gibt
allerdings eine Warnung bei Gleitkommazahl -> Integer Konvertierung.

    int i = 3 + 7.89;                       // Warnung, i = 10

Wenn man die Konvertierung möchte, kann man "casten":

    int i = int (3 + 7.89);                 // i = 10
    int i = (int) (3 + 7.89);               // i = 10
    int i = static_cast <int> (3 + 7.89);   // i = 10

================================================================================
Aufgabe 1: Strukturen
================================================================================
(a) Ergänze das Bruchprogramm um kürzen und die anderen Grundrechenarten.

(b) Stell Dir vor, Du muesstest einen PDA entwerfen. Welche Strukturen würdest
    Du vorschlagen, um die Daten des Nutzers zu speichern?

================================================================================
Aufgabe 2: doubles
================================================================================
(a) Schreibe eine Funktion

    double drand();

    welche (möglichst gleichverteilte) Zufallszahlen im Intervall [0,1)
    zurückgibt.

(b) Schreibe ein Programm, was PI nach der Monte-Carlo-Methode berechnet. Diese
    ist:

    1. man nimmt N zufällige (x,y) Koordinaten, mit
       0 <= x < 1
       0 <= y < 1
    2. man zählt, wieviele (K) davon im Einheitskreis liegen
    3. aus dem Quotient: K/N schliesst man auf PI

(c) Erweitere wurzel2() so, dass es beliebige Wurzeln berechnen kann.
```