Algorithmen und Programmierung

2. Kapitel
Einführung in Programmiersprachen

Prof. Matthias Werner
Professur Betriebssysteme

2.1 Grundsätzliches

- Die Ausführung eines Algorithmus auf einem Computer setzt voraus, dass der Computer den Algorithmus **interpretieren** kann:
  - Er muss „verstehen“, was jeder Schritt bedeutet
  - Er muss die jeweilige Operation ausführen können
- Jeder Prozessor[1] kann ein Bitmuster interpretieren
  - Dieses Bitmuster wird **Maschinencode** genannt
  - Der Maschinencode ist typisch für jede Prozessorfamilie
- Beispiel: Im Maschinencode der x86-Prozessorfamilie bedeutet das Bitmuster

  `11111010`

  dass der Prozessor sich nicht „ablenken“ lassen soll, d.h., dass keine Interrupts[2] beachtet werden sollen

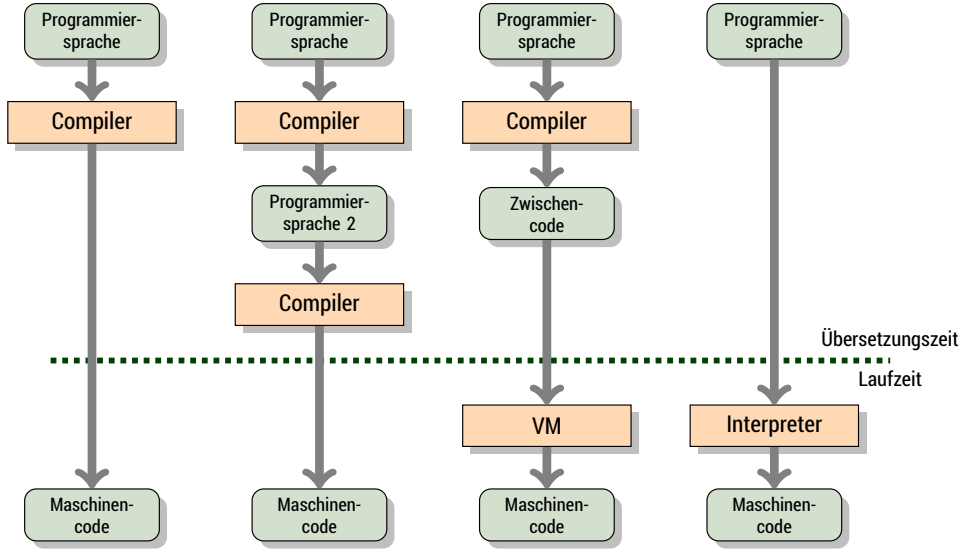
[1] Präziser: jeder **digitale** Prozessor
[2] Das Konzept des Interrupts wird in den Fächern der technischen Informatik genauer behandelt

Compiler und Interpreter

- Für den Menschen ist Maschinencode nicht besonders „handlich“ ☺
- Daher gibt es Programmiersprachen, die dem Denken des Menschen entgegenkommen
- Programmiersprachen benutzen verschiedene Modelle und Abstraktionen
  - Z.B. gibt es für jeden Maschinencode (mindestens) eine sogenannte **Assembler-Sprache**, die den Maschinencode in leicht(er) merkbaren Abkürzungen (**Mnemonics**) darstellt

    `11111010` ➡ `cli` (für **cl**ear **i**nterrupt)
- Programme werden entweder in Maschinencode **übersetzt**, oder durch ein anderes Programm **interpretiert**
  - Übersetzer = **Compiler**
  - **Interpreter**

Varianten der Programmausführung



Paradigmen

- Häufig werden Sie hören, dass eine Programmiersprache diesem oder jenem **Paradigma** folgt

Paradigma

Paradigma, das, Pl: Paradigmen, griech., $\pi\alpha\rho\alpha\delta\varepsilon\iota\gamma\mu\alpha$: Beispiel, Muster

- Gemeint ist, dass die Sprache gewisse **Eigenschaften** hat oder bestimmte **Programmierstile** (besonders gut) unterstützt
- Typischerweise unterstützen Sprachen mehrere Paradigmen
- Die Befolgung eines Paradigma sagt **nichts** über die Ausdrucksmächtigkeit einer Sprache

Paradigmen (Forts.)

Eigenschaft	(vereinfachte) Erläuterung	typische Beispiele
imperativ	betont das Zustandsübergangsmodell der Programmausführung	C, C++, C#, Java, Pascal
funktional	betont das Funktionsmodell der Programmausführung	Haskell, Scheme, Logo, ML
deklarativ	ermöglicht die Beschreibung eines Programm**ziels**, statt des Weges dorthin	SQL, Prolog
modular	ein Programm kann aus verschiedenen Teilen „komponiert“ werden, wobei von den externen Teilen nur bestimmte Informationen bekannt werden	Modula 2, C#,
objektorientiert	betont die Zusammengehörigkeit von Daten und Operationen	Simula, Smalltalk, Eiffel, Java, C++
strikt	Ausdrücke werden bei Zuweisung berechnet	C, Java, Ada
nicht-strikt	Ausdrücke werden bei Bedarf berechnet	Miranda, Haskell
reflexiv	Programm kann sich selbst modifizieren	Lisp, C#, Python

Paradigmen (Forts.)

- Prinzipiell kann **jedes** lösbare Problem in **jeder Sprache** gelöst werden
- Die Befolgung von Paradigmen können häufig auch durch Selbstbeschränkung oder Tricks erreicht werden, wenn die Sprache sie nicht unterstützt
- **Beispiele**: Die Programmiersprache C unterstützt keine Objektorientierung und C++ ist keine funktionale Sprache.
  Es ist aber sehr wohl möglich, in C objektorientiert oder in C++ funktional zu programmieren
  - Objektorientierung in C: siehe z.B. [Sch01]
  - Funktionales Programmieren in C++ siehe z.B.: [MS00]

2.2 Geschichte

Erste Computer

1822 Erster funktionsfähiger Entwurf einer **Analytical Engine** durch CHARLES BABBAGE; besitzt alle Einheiten eines modernen Rechners wie Rechenwerk, Zahlenspeicher, Steuerwerk, E/A– Möglichkeiten;
Programmentwurf von AUGUSTA ADA KING BYRON zur Berechnung der Bernoulli-Zahlen

1936 ALAN M. TURING entwirft die sogenannte „Turing-Maschine“

1941 KONRAD ZUSE entwickelt den ersten funktionsfähigen Computer **Z3**

1944 **ENIAC** (Electrical Numerical Integrator And Computer): erster elektronischer Universalrechner

Computer wurden in dieser Zeit entweder durch „Verkabelung“ oder mit Lockkarten/-steifen in Maschinencode programmiert

1952 GRACE HOPPER entwickelt den ersten Compiler (Assembler)

Höhere Programmiersprachen

~1946 KONRAD ZUSE entwirft **Plankalkül**, die erste vermutlich erste höhere Programmiersprache; erste Implementierung 1972

1953 **FORTRAN** (**For**mula **Tran**slating System): erste implementierte höhere Programmiersprache von JOHN W. BACKUS
- Viele bis heute verwendete Konzepte, Stammvater aller imperativen Sprachen
- Insbesondere für numerische Berechnungen vorgesehen und optimiert
- Immer wieder aktualisiert, letzte Version Fortran 2008

1958 **Lisp** (**List** **P**rocessing), entwickelt von JOHN MCCARTHY am MIT
- Bis heute in Verwendung, mit vielen Abkömmlingen, Stammvater aller funktionalen Sprachen
- Angelehnt an das λ-Kalkül
- Symbolische Programmierung
- Funktionen können auch lediglich teilweise ausgewertet und als Parameter übergeben werden
- automatische Speicherverwaltung (*garbage collector*)
- Mechanismen zur Robustheit (statische Zusicherungen, Alignment, Grenzüberprüfungen)

Höhere Programmiersprachen (Forts.)

1958 **ALGOL** (**Algo**rithmic **L**anguage)
- Vorläufer vieler Programmiersprachen, z.B. C, Pascal, Java, ...
- allgemeine Schleifen
- Blockkonzept

1962 **Simula** (**Simu**lation **la**nguage) entwickelt von OLE-JOHAN DAHL und KRISTEN NYGAARD
- Idee der Objektorientierung
- Vorgesehen für den Entwurf von Simulationssoftware
- Konzept von Co-Routinen

1970 **Smalltalk** von ALAN KAY, DAN INGALLS und ADELE GOLDBERG
- Erste vollständig objektorientierte Sprache: alles ist ein Objekt
- Virtuelle Maschine (VM)

1972 **Prolog** (**Pro**grammation en **Log**ique) von ALAIN COLMERAUER
- Deklarative Programmiersprache zur Beschreibung logischer Sachverhalte
- Hauptanwendungszweck: Künstliche Intelligenz

Höhere Programmiersprachen (Forts.)

Man beachte:

Damit wurden im Prinzip alle wesentlichen Programmiersprachkonzepte schon in den frühen Sprachen etabliert.

- Bis zum heutigen Tag wurden über 2500 Computersprachen entwickelt
- `http://people.eecs.ku.edu/~kinners/new_general.html` listet viele davon auf
- Im Anhang des Skripts sind für die wichtigsten 50 Sprachen Erscheinungszeiten und Abhängigkeiten in einem Graphen dargestellt

Die Programmiersprache C

Etwa 1972 entstand die erste Version von **C**
- Entwickelt von BRIAN W. KERNIGHAN und DENNIS RITCHIE
- Der Erfolg von C ist eng mit dem Erfolg des Betriebssystems **UNIX** verknüpft
  - Die ersten Versionen von UNIX waren in Assembler entwickelt
  - 1973 wurde UNIX nahezu komplett in C reimplementiert, was maßgeblich zum Erfolg beitrug
- Vorgängersprachen: **BCPL** ➡ **B** ➡ **C**
- Eine Vielzahl von Sprachen stammen von C ab oder wurden durch C inspiriert, u.a.: **C++**, **C***, **Concurrent C**, **C+@**, **Objective C**, **Cmm**, **Java**, **C#**, **Python**, ...

Die Programmiersprache C (Forts.)

Die Sprache war im Laufe ihrer Geschichte vielen Wandlungen unterworfen

- **K&R C**: Erste (öffentliche) Version von C: 1978
- 1983 begann das ANSI-Komitee **X3J11** die Sprache zu standardisieren.
- 1989 wurde der Standard ANSI X3.159-1989 verabschiedet, bekannt als **C89** oder **ANSI C**
  - U.a. geänderte Funktionsdeklarationen
- 1990 wurde der Standard von der ISO als ISO/IEC 9899:1990 (oder **C90**) übernommen
- Weiterentwicklung des Standards führten zu **C95** (ISO/IEC 9899:1994-09), das nur kleinere Änderungen gegenüber C90 einführte

Die Programmiersprache C (Forts.)

- Größere Änderungen gab es mit ISO/IEC 9899:1999, bekannt als **C99**
  - Änderungen bei Typen, u.a. optionale Typen mit festgelegter (Mindest-)breite
  - Felder variabler Größe
  - Mehr Freiheiten bei der Deklaration
  - Schlüsselwörter für Optimierung
  - Kommentare von von C++ übernommen
  - Verbot von impliziten Typen oder Funktionsdeklarationen
  - Später kleinere Änderungen in *Technical Corrigenda*

Die Programmiersprache C (Forts.)

- Aktueller Standard: **C11** (Dezember 2011, ISO/IEC 9899:2011)
  - Explizites Maschinen- und Speichermodell
  - Bessere Unterstützung von Unicode
  - Viele Änderungen an der Standardbibliothek, z.B. Unterstützung von Multithreading
  - Unterstützung von Typengenerizität in Makros
  - Von den meisten Compilern derzeit nicht vollständig implementiert

Die Programmiersprache C (Forts.)

- Wir werden in diesem Kurs i.d.R. vom C99-Standard ausgehen, ohne alle Möglichkeiten von C99 auszuschöpfen
- Viele Compiler unterstützen mehrere Standard
  - Gewünschter Standard muss ggf. dem Compiler mitgeteilt werden
- Wo es wesentliche Unterschiede gibt, wird im Skript darauf hingewiesen: 

Achtung

Einige Standards, insbesondere C11, werden von verschiedenen Compilern nicht vollständig unterstützt, was zu merkwürdigen Effekten führen kann.
Versuchen Sie, möglichst wenig versionsabhängig zu programmieren.

Eigenschaften von C

- Ist nahe an der **Hardware**
  ➡ manchmal wird nicht von einer Hochsprache, sondern von einer „Mittelsprache“ gesprochen
  ➡ Alle relevanten Betriebssysteme sind in C geschrieben
- Programm sind sehr **kompakt** ➡ sie brauchen geringen Overhead und Speicher
- Ist weit **verbreitet**
  ➡ vermutlich immer noch die Sprache, in der weltweit die meisten Programme geschrieben werden
- In C lassen sich sehr **effiziente** Programme schreiben
  ➡ Geschwindigkeit von C-Programmen ist meist Maßstab

Größter Vorteil von C

C gibt dem Programmierer viele Freiheiten.

Größter Nachteil von C

C gibt dem Programmierer viele Freiheiten.

Geschichte von Python

- Da in diesem Kurs gelegentlich auch Python genutzt wird,gibt es auch davon eine (eher kurze) geschichtliche Darstellung Rossum]van Rossum, Guido
  - Erste Implementation von GUIDO VAN ROSSUM (Centrum Wiskunde & Informatica, Amsterdam) im Jahre 1989
  - Name abgeleitet von Monty Python's Flying Circus
  - 2000 erschien Version 2.0 und wurde zu einem OSS-Projekt
  - Nach mehreren Zwischenversionen erschien 2008 die **nicht** rückwärtskompatible Version 3.0
  - Aktuell: 3.5.2 (Juni 2016), jedoch wird 2.7 weiter gepflegt (aktuell: 2.7.12)

- Python hat trotz kurzer Geschichte schon einige andere Sprachen beeinflusst, u.a.: **Cobra**, **ECMAScript**, **Go**, **Groovy**

2.3 C am Beispiel

Warnung

Bevor wir mit dem Programmieren in C beginnen, ein Wort der Warnung:

Achtung!

C-Compiler (insbesondere ältere) gehen davon aus, dass Programmierer wissen, was sie wollen.
Viele Fehler treten dann erst zur Laufzeit auf.

- Beispiel: Der folgende Code stellt ein übersetz- und ausführbares C-Programm dar...

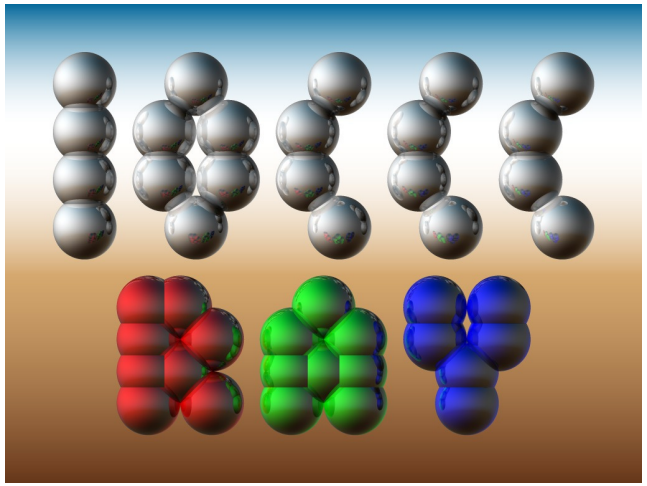
Warnung (Forts.)

```
X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMMOO055557799@@>>>BBBGGIIKK"[b]-64;C="C@=::C@@==@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b ]-64;S=b<22?9:0;D=2;}I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,  I (x,Y/2,X
)):(E=X);      }H(x){I(x,    _,0);}p;q(         c,x,y,z,k,l,m,a,          b){F(c
);x-=E*M     ;y-=S*M           ;z-=C*M          ;b=x*        x/M+          y*y/M+z
*z/M-D*D    *M;a=-x             *k/M     -y*l/M-z        *m/M;    p=((b=a*a/M-
b)>=0?(I    (b*M,_       ,0),b    =E,      a+(a>b      ?-b:b)):      -1.0);}Z;W;o
(c,x,y,     z,k,l,    m,a){Z=!    c?      -1:Z;c     <44?(q(c,x          ,y,z,k,
l,m,0,0     ),(p>      0&&c!=     a&&        (p<W          ||Z<0)          )?(W=
p,Z=c):     0,o(c+          1,    x,y,z,         k,l,          m,a)):0     ;}Q;T;
U;u;v;w    ;n(e,f,g,           h,i,j,d,a,    b,V){o(0      ,e,f,g,h,i,j,a);d>0
&&Z>=0? (e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h-=u*E/(M/2),i-=v*E/(M/2),j-=w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,        U/=2,V=V<22?7:  (V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T                +=V&2?b        :0,U+=V    &4?b:0)     :(d==P?(g+=2
,j=g>0?g/8:g/      20):0,j    >0?(U=      j    *j/M,Q      =255-    250*U/M,T=255
-150*U/M,U=255    -100    *U/M):(U    =j*j     /M,U<M           /5?(Q=255-210*U
/M,T=255-435*U           /M,U=255    -720*      U/M):(U       -=M/5,Q=213-110*U
/M,T=168-113*U    /       M,U=111               -85*U/M)      ),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<    0?0:      Q>0?     0:          Q;T=T<0?    0:T>0?0:T;U=U<0?0:
U>0?0:U;}R;G;B    ;t(x,y     ,a,    b){n(M*J+M    *40*(A*x   +a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,  -1,0,0);R+=Q    ;G+=T;B   +=U;++a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}
```

Warnung (Forts.)

- ... das folgendes Bild generiert:



- Mehr davon beim **International Obfuscated C Code Contest**:
  `http://www.de.ioccc.org/main.html`

Programmentwicklung

- Wir werden uns in diesem Kurs auf die Programmierung in C konzentrieren, aber gelegentlich auch Python benutzen
- Die Programmerstellung folgt stets gleichem Arbeitsablauf:

Schritt 1: Programmcode bearbeiten
Schritt 2: Programm übersetzen und linken
- Falls dabei Fehler auftreten, zurück zu Schritt Schritt 1:

Schritt 3: Programm testen
- Falls dabei Fehler auftreten, zurück zu Schritt Schritt 1:

Anmerkung:

Es wird davon ausgegangen, dass jeder Kursteilnehmer in der Lage ist, mit einer Shell (tcsh, bash, ...) soweit umzugehen, dass sie/er ein Verzeichnis anlegen/wechseln/löschen, mit einem Editor Textdateien bearbeiten und allgemein Befehle in eine Shell eingeben kann.

Werkzeuge

- Für die C-Programmerstellung braucht man mehrere Werkzeuge, mindestens zwei: einen Editor und einen Compiler
- Als Editor kann jedes Programm genutzt werden, dass **einfachen** (unformatierten) Text bearbeiten kann
  - Editoren auf Shellebene: vim, emacs, nano, mcedit, ...
  - Grafische Editoren: kedit, kate, xemacs, ...
  - Gänzlich ungeeignet: ~~MS Word~~, ~~OpenOffice~~, ~~kWriter~~, ...

Hinweis

Benutzen Sie den Editor, der Ihnen am besten liegt.
Wenn Sie aber einen Editor einer IDE (z.B. Eclipse) benutzen, achten Sie darauf, dass Sie **nicht** die automatische Übersetzung der IDE benutzen

Werkzeuge (Forts.)

- Als Compiler werden wir stets den **clang**-Compiler oder den Compiler der **GNU Compiler Collection**[3] nutzen
  - Beide sind auf jeder relevanten Plattform (Linux, MacOS, Windows, FreeBSD, ...) verfügbar
  - Prinzipiell sind jedoch auch andere C-Compiler möglich

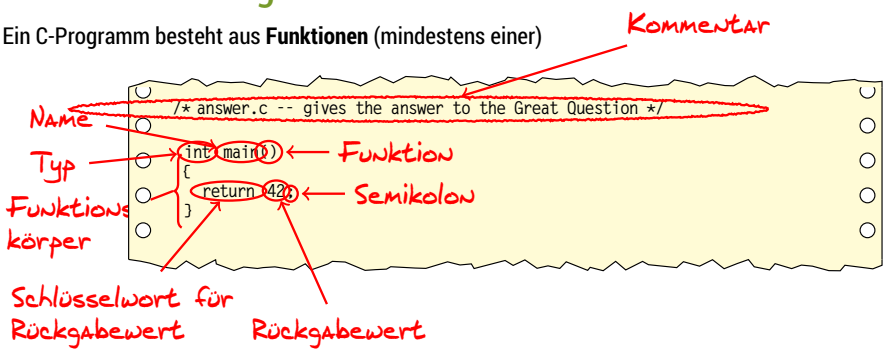
Hinweis

Auch wenn dies denjenigen unter Ihnen, die schon etwas Programmiererfahrung haben, vermutlich ungewohnt vorkommt:
Wir werden **den Compiler zunächst stets von der Kommandozeile** (oder über ein `Makefile`, siehe Kapitel **??**) starten.
Ziel ist (zunächst) nicht die Programmier-Bequemlichkeit, sondern dass Sie verstehen, was Sie genau machen.

[3] Die Kommandozeilenoptionen sind für `clang` und GNU-Compiler in den meisten Fällen gleich.

Das erste C-Programm

Ein C-Programm besteht aus **Funktionen** (mindestens einer)



- **Kommentar:** wird bei der Programmausführung ignoriert, ist aber wichtig für das Verständnis des Programms
- **Typ:** Wertebereich der Funktion; `int` steht für „*Integer*“ ≙ ganze Zahl
- Jede Funktion hat einen **Namen**; „main“ ist die „Startfunktion“
- Die runden Klammer kennzeichnen „main“ als **Funktion**
- Der **Funktionskörper** (oder Funktionsrumpf) wird in geschweifte Klammern eingeschlossen
- Das **Schlüsselwort** „return“ kündigt den Rückgabewert an
- Der Wert „42“ wird von der Funktion zurückgegeben ➡ **Rückgabewert**
- Das Semikolon markiert das Ende einer **Anweisung** (Statement)

Übersetzen eines C-Programms

Um das Programm zu übersetzen (compilieren) wird der Compiler aufgerufen:

```
> cc -std=c99 -Wall answer.c -o answer
```

- `cc`: Der Name des Compilers
- `-std=c99`: Aktiviert den Sprachstandard C99
- `-Wall`: Gibt Warnungen aus
- `answer.c`: Name des Quellcode-Files
- `-o answer`: Bestimmt den Namen des übersetzten Programms

Aufruf des Programms:

```
> ./answer
> echo $?
42
>
```


Der erste Algorithmus

- Das folgende Programm implementiert eine Variante des Euklidschen Algorithmus (siehe Beispielsalgorithmus 6)
- Zur (hoffentlich!) besseren Übersichtlichkeit, enthält das Listing Zeilennummern und farbige **Schlüsselwörter**

```
1  /* euclid.c -- calculates GCD */
2
3  // announce function to compiler
4  static int euclid(int, int);
5
6  int main ()
7  {
8    return euclid (45 ,30);
9  }
10
11 // defines the function
12 int euclid(int x, int y)
13 {
14   if (x==y) return x;
15   else if (x>y) return euclid(x-y,y);
16   else return euclid(y-x,x);
17 }
```


Der erste Algorithmus (Forts.)

Betrachten Code genauer:

#2: Macht den Compiler mit einer neuen Funktion bekannt ➡ **Deklaration**

- Schlüsselwort `static` = Funktion ist lokal (wird nicht an andere Module exportiert)
- `int euclid(int, int);`: Funktion hat den Namen „euclid“, hat zwei Ganzzahlen als Argumente und gibt eine Ganzzahl zurück

#7: Der Rückgabewert ist Wert der Funktion „euclid“ mit den Parametern 45 und 30

#10: Hier beginnt die **Definition** der Funktion

- Die Parameter bekommen **innerhalb** der Funktion die Namen „x“ und „y“

#12 - 14: Der eigentlich Algorithmus

- Die Schlüsselworte „`if`“ und „`else`“ steuern die bedingte Ausführung

Übersetzen und aufrufen:

```
> cc -std=c99 -Wall euclid.c -o euclid
>
> ./euclid
> echo $?
15
>
```

Schlüsselwörter

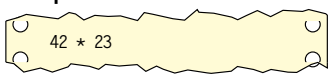
- Für den Algorithmus haben wir jetzt schon fünf Schlüsselwörter benutzt: `static`, `int`, `return`, `if` und `else`
- Während `static` und `int` dem Compiler etwas erklären, steuern die anderen den **Programmfluss**, also *was* gemacht wird
- `return` haben wir schon betrachtet: Es gibt einen Wert an, die eine Funktion liefert und beendet diese Funktion
  - Im Fall der Hauptfunktion `main()` wird damit auch das gesamte Programm beendet
- `if` und `else` sind für die **bedingte Ausführung** zuständig
  - `if` untersucht eine Bedingung (hier: ob `x` und `y` gleich sind) und führt etwas aus, wenn die Bedingung eingetroffen ist (*wahr ist*)
  - `else` erklärt die **Alternative**: Es sagt, was ausgeführt wird, wenn die Bedingung **nicht** eingetroffen ist
  - `if` und `else` gehören zusammen: es gibt **nie** ein `else` ohne ein vorheriges `if`
  - Allerdings braucht ein `if` nicht ein folgendes `else` ➡ dann wird als Alternative einfach nichts gemacht

Operatoren

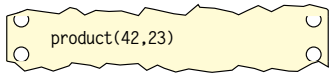
- In unserem C-Programm haben wir **Operatoren** benutzt
  - Aus der Mathematik bekannt und funktionieren analog
- Beispiel: Minus-Operator („`-`“) bildet Differenz zweier Zahlen: `x - y`
- Andere Operatoren in C
  - `+` – bildet Summe zweier Zahlen
  - `*` – bildet Produkt zweier Zahlen
  - `/` – bildet Quotienten zweier Zahlen
  - `%` – bildet Rest der Ganzzahldivision zweier Zahlen (Modulo)
  - `<`, `>`, `<=`, `>=`, `==` – Vergleichsoperatoren „kleiner“, „größer“, „kleiner/gleich“, „größer/gleich“, „gleich“
- Operatoren, die zwischen zwei Ausdrücken stehen, werden **Infixoperatoren** genannt
- Es gibt noch viele andere Operatoren, die wir im Laufe dieses Kurses kennenlernen werden
  - In manchen Programmiersprachen kann man sich **eigene** Operatoren definieren – in C geht dies nicht

Operatoren (Forts.)

- Operatoren können als eine spezielle Art angesehen werden, Funktionen zu beschreiben
- Beispielsweise könnte statt



auch



stehen

- Die Operatorenschreibweise lässt sich aber besser lesen 🙂
- Operatoren können auch aus mehr als einem Zeichen bestehen
  - Beispielsweise dient der **Klammeroperator** `( ... )` dazu, alles was zwischen der öffnenden und schliessenden Klammer steht, zu einer Gruppe zusammenzufassen
- Je nachdem, woher ein Operator seine Operanden „bezieht“, unterscheidet man
  - **Präfixoperator** – steht **vor** Operand(en)
  - **Postfixoperator** – steht **nach** Operand(en)
  - **Infixoperator** – steht **zwischen** Operanden
- Z.B. ist in C die Multiplikation „`*`“ ein Infixoperator

Rekursion – Modell der kleinen Menschen

- Man nehme an, dass Berechnungen nicht von einem Computer durchgeführt werden, sondern von einer Gruppe tausender fleissiger kleiner Menschen.[4]
- Den **Algorithmus** (Anleitung) kennen alle, bei der **Durchführung** werden die Aufgaben verteilt.
- **Beispiel „Euklidscher Algorithmus“**:
  - **Manfred** ist für die Funktion `main` verantwortlich. Dazu geht er zu **Ernst**, und sagt ihm die Werte 45 und 30
  - **Ernst** (für `euclid` verantwortlich) vergleicht die Zahlen, die **Manfred** ihm gegeben hat. Da sie ungleich sind, zieht er die kleinere von der größeren ab ($45 - 30 = 15$) und sagt nun **Emil** die Werte 15 und 30
  - **Emil** (ebenfalls für `euclid` verantwortlich) vergleicht die Zahlen, die **Ernst** ihm gegeben hat. Da sie ungleich sind, zieht er die kleinere von der größeren ab ($30 - 15 = 15$) und sagt nun **Egon** die Werte 15 und 15
  - **Egon** (ebenfalls für `euclid` verantwortlich) vergleicht die Zahlen, die **Emil** ihm gegeben hat. Da sie gleich sind, gibt er **Emil** den Wert 15 zurück
  - **Emil** gibt den Wert **Ernst**, **Ernst** gibt ihn **Manfred**, und **Manfred** teilt ihn der Öffentlichkeit mit

[4] Dieses Modell geht auf Brian Harvey zurück, siehe z.B. [HW94]; [Har97]

Rekursion – Modell der kleinen Menschen (Forts.)

- Das Beispiel soll folgendes verdeutlichen:
  - Jeder der kleinen Menschen weiß nicht, was der andere für Werte hat. Z.B. Egon hat **keine Ahnung**, welche Zahlen Ernst erhalten hat
  - Obwohl die Zahlen in der Gebrauchsanweisung immer x und y heißen, sind es für jeden der kleinen Menschen **unterschiedliche** Zahlen
  - Obwohl keiner der kleinen Menschen weiß, mit welchen Zahlen ein anderer arbeitet, **funktioniert** der Algorithmus

Programm mit Ein- und Ausgabe

- Zur Ein- und Ausgabe werden **Bibliotheksfunktionen** genutzt
- An `main()` werden Parameter der Befehlszeile übergeben

```c
/* euclid2.c -- calculates GCD with I/O */

extern int printf(const char*,...); /* output   */
extern int atoi(const char*);       /* text -> integer */
static int euclid(int,int);

int main(const int c, const char* v[])
{
  printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
  return 0;
}

int euclid(int x, int y) {
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```


Programm mit Ein- und Ausgabe (Forts.)

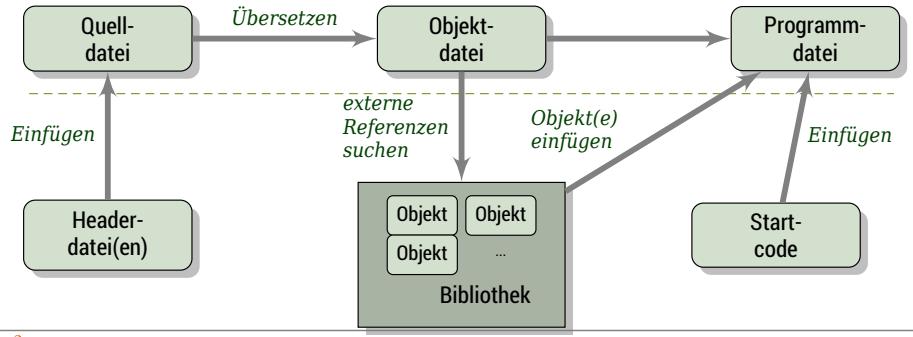
- Wir haben mit `atoi()` und `printf()` unsere erste Bibliotheksfunktionen genutzt
  - `atoi()` steht für „ascii to integer“ und dient dazu, die Kommandozeilenparameter in Zahlen zu verwandeln
  - `printf()` gibt (formatierten) Text auf dem Bildschirm aus und wandelt dazu Zahlen in Text um
- Die Kenntnis von Bibliotheksfunktionen ist sehr hilfreich beim Programmieren
- Die genaue Funktionsweise der Funktionen werden wir erörtern, wenn wir Typen (siehe Kapitel **??**) diskutiert haben
- Generell kann man die Beschreibung einer Funktion (in UNIX-artigen) Betriebssystemen mit dem `man`-Befehl nachschlagen

```
ATOI(3)                 BSD Library Functions Manual                 ATOI(3)
BEZEICHNUNG
     atoi, atol, atoll, atoq - konvertiert eine Zeichenkette in eine Integer-Zahl

ÜBERSICHT
     #include <stdlib.h>
       :
       :
```


Bibliotheken und Linker

- Im letzen Beispiel wurden zwei Funktionen benutzt, die **nicht Bestandteil** der Sprache C sind
- Sie wurden von anderen Leuten in C geschrieben (➡ Schlüsselwort `extern`) und in der **Standard-C-Bibliothek** zur Verfügung gestellt
  - Die Standard-C-Bibliothek ist ebenfalls genormt
  - Jeder kann Bibliotheken schreiben ➡ Wiederverwendung von Code
  - Deklarationen zu Bibliotheken sind in **Headerdateien** zusammengefasst
  - Headerdateien werden über `#include`-Direktive in Code eingefügt



Bibliotheken und Linker (Forts.)

- Unter Nutzung der Headerdateien sieht unser Programm jetzt so aus:

```c
/* euclid3.c -- calculates GCD with I/O */
#include<stdio.h>  /* printf */
#include<stdlib.h> /* atoi   */

static int euclid(int,int);

int main(const int c, const char* v[])
{
  printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
  return 0;
}

int euclid(int x, int y) {
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```


Blick unter die Haube

- Die verschiedenen Schritte bei der Programmerstellung eines C-Programms macht der Compiler nicht allein
- Vielmehr werden eine Reihe anderer Programme aufgerufen
- Im einzelnen passiert folgendes:

Weg zum ausführbaren Programm

1. Der **Präprozessor** wertet Direktiven aus und fügt Inhalt von Headerdateien in den Code ein
2. Der eigentliche **Compiler** übersetzt den C-Code in Assembler-Code
3. Der **Assembler** übersetzt den Assembler-Code in Maschinencode und generiert daraus eine Objektdatei
4. Der **Linker** sucht die externen Referenzen des Objektfiles und sucht entsprechende Objekte aus der **Standardbibliothek** oder anderen angegebenen Bibliotheken (und Start-Objekt) zusammen und bindet alles zu einer ausführbaren **Programmdatei**

Blick unter die Haube (Forts.)

- Man kann den Compiler dazu veranlasst werden, nur einzelne Stufen des Erstellungsprozesses durchzuführen
  - z.B.: „`-E`“ stoppt nach dem Präprozessor
- Beim GNU-Compiler kann man mit Hilfe der Option „`-v`“ gut sehen, was im einzelnen passiert (*das folgende Beispiel ist gekürzt*)

```
> gcc -ansi -pedantic -Wall -v euclid3.c -o euclid3
Using built-in specs.d
Target: i686-apple-darwin10
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/cc1 -quiet -v -imultilib x86_64 -
 D__DYNAMIC__ euclid3.c -fPIC -quiet -dumpbase euclid3.c -mmacosx-version-min=10.6.4
  -m64 -mtune=core2 -ansi -auxbase euclid3 -pedantic -Wall -ansi -version -o /var/
 folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//ccZlbYNd.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/as -arch x86_64 -force_cpusubtype_ALL -o
 /var/folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//cc65KfC6.o /var/folders/WI/
 WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//ccZlbYNd.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/collect2-dynamic -arch x86_64 -
 macosx_version_min 10.6.4 -weak_reference_mismatches non-weak -o euclid3 -lcrt1
 .10.6.o -L/usr/lib/gcc/i686-apple-darwin10/4.2.1/x86_64 -L/usr/lib/gcc/i686-apple-
 darwin10/4.2.1/x86_64  -L/usr/lib/gcc/i686-apple-darwin10/4.2.1 -L/usr/lib/gcc/i686
 -apple-darwin10/4.2.1/../../.. /var/folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//
 cc65KfC6.o -lSystem -lgcc -lSystem
```


Blick unter die Haube (Forts.)

- Man kann den Compiler dazu veranlasst werden, nur einzelne Stufen des Erstellungsprozesses durchzuführen
  - z.B.: „`-E`“ stoppt nach dem Präprozessor
- Beim GNU-Compiler kann man mit Hilfe der Option „`-v`“ gut sehen, was im einzelnen passiert (*das folgende Beispiel ist gekürzt*)

```
> gcc -ansi -pedantic -Wall -v euclid3.c -o euclid3
Using built-in specs.d
Target: i686-apple-darwin10
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/cc1 -quiet -v -imultilib x86_64 -
 D__DYNAMIC__ euclid3.c -fPIC -quiet -dumpbase euclid3.c -mmacosx-version-min=10.6.4
  -m64 -mtune=core2 -ansi -auxbase euclid3 -pedantic -Wall -ansi -version -o /var/
 folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//ccZlbYNd.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/as -arch x86_64 -force_cpusubtype_ALL -o
 /var/folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//cc65KfC6.o /var/folders/WI/
 WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//ccZlbYNd.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/collect2-dynamic -arch x86_64 -
 macosx_version_min 10.6.4 -weak_reference_mismatches non-weak -o euclid3 -lcrt1
 .10.6.o -L/usr/lib/gcc/i686-apple-darwin10/4.2.1/x86_64 -L/usr/lib/gcc/i686-apple-
 darwin10/4.2.1/x86_64  -L/usr/lib/gcc/i686-apple-darwin10/4.2.1 -L/usr/lib/gcc/i686
 -apple-darwin10/4.2.1/../../.. /var/folders/WI/WIOtIF54VPPoSkTWoc+++-NM/-Tmp-//
 cc65KfC6.o -lSystem -lgcc -lSystem
```

Compiler
Assembler
Linker

2.4 Python

Das Gleiche in Blau/Gelb

- Python wird zur Laufzeit interpretiert ➡ **Skriptsprache**
- Das Great-Answer-Programm sieht in Python so aus:

```
# answer.py -- gives the answer to the Great Question

exit(42)
```

- Zur Ausführung muss der Interpreter, also das Programm, das den Code „versteht“, angegeben werden:

```
> python answer.py
> echo $?
42
```


Das Gleiche in Blau/Gelb (Forts.)

- Unter UNIX (z.B. Linux) kann alternativ auch in der 1. Zeile der Interpreter angegeben werden:

```
#! /usr/bin/env python
# answer2.py -- gives the answer to the Great Question

exit(42)
```

```
> ./answer.py
> echo $?
42
```

- Außerdem muss die Datei ausführbar gemacht werden, z.B.: `chmod a+x answer.py`.

Euklid

- Der Euklid-Algorithmus kann hier ähnlich wie in C beschrieben werden
- Es gibt in Python keinen Startpunkt, sondern das Programm wird von oben nach unten abgearbeitet

```
# euclid.py -- calculates GCD

def euclid(x,y):
    if x==y: return x
    elif x>y: return euclid(x-y,y)
    else: return euclid(y-x,x)

exit(euclid(45,30))
```

Achtung!

Während bei C Einrückungen nur der besseren Lesbarkeit dienen, sind sie bei Python verpflichtend.

Interaktivität

- Python kann interaktiv benutzt werden
- Dazu wird das `exit`-Statement aus dem Skript gelöscht

```
# euclid2.py -- calculates GCD

def euclid(x,y):
    if x==y: return x
    elif x>y: return euclid(x-y,y)
    else: return euclid(y-x,x)
```

```
> ./python
Python 2.7 (r27:82500, Jul 17 2010, 23:39:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> from euclid2 import euclid
>>> euclid(30,45)
15
>>> euclid(42,23)
1
>>>
```

- Die Datei „euclid2.py“ bildet ein **Modul**, das die Funktion der Bibliothek in C übernimmt

Zwischencode

- Nach den letzten Beispielen wird man im Arbeitsverzeichnis die Datei „euclid2.pyc“ finden
- Dies ist der Zwischencode (Bytecode), in den Python das Programm übersetzt, bevor es interpretiert wird
- Bei einem erneuten Aufruf wird – falls die Quelldatei nicht geändert wurde – direkt der Zwischencode ausgeführt
- Man kann ein Skript „xyz.py“ übersetzen, ohne es auszuführen:

```
>>> import py_compile
>>> py_compile.compile('xyz.py')
```

- Alternativ können alle Python-Dateien eines Verzeichnisses übersetzen:

```
> ls
answer.py euclid.py euclid2.py euclid3.py
> python -mcompileall .
Listing . ...
Compiling ./answer.py ...
Compiling ./euclid.py ...
Compiling ./euclid2.py ...
Compiling ./euclid3.py ...
> ls
answer.py euclid.py euclid2.py euclid3.py
answer.pyc euclid.pyc euclid2.pyc euclid3.pyc
```


Ein- und Ausgabe

- Es gibt jede Menge vordefinierte Module, z.B. „sys“, welches ein Interface zum Betriebssystem bereitstellt

```
# euclid3.py -- calculates GCD
from sys import argv

def euclid(x,y):
    if x==y: return x
    elif x>y: return euclid(x-y,y)
    else: return euclid(y-x,x)

print(('GCD is', euclid(int(argv[1]),int(argv[2]))))
```


2.5 Fehler

```
/* euclid-fail.c -- a faulty euclid */

// announce function to compiler
static int euclid(int; int);

int main()
{
   return euclid(45,30);
}

// defines the function
int euclid(int x, int y)
{
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```

```
> gcc -std=c99 -Wall euclid.c -o
      euclid
euclid.c:4:1: error: parameter '({
      anonymous})' has just a forward
       declaration
euclid.c: In function 'main':
euclid.c:8:10: error: too many
      arguments to function 'euclid'
euclid.c:4:12: note: declared here
euclid.c: At top level:
euclid.c:12:5: error: conflicting
      types for 'euclid'
euclid.c:4:12: note: previous
      declaration of 'euclid' was
      here
euclid.c:4:12: warning: 'euclid'
      used but never defined
>
```


Und noch einmal...

```
/* euclid-fail2.c -- again faulty
*/

// announce function to compiler
static int euclid(int, int);

int main()
{
   return euklid(45,30);
}

// defines the function
int euclid(int x, int y)
{
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```

```
> gcc -std=c99 -Wall euclid.c -o
      euclid
euclid.c: In function 'main':
euclid.c:8:10: warning: implicit
      declaration of function 'euklid
      ' [-Wimplicit-function-
      declaration]
euclid.c: At top level:
euclid.c:12:5: warning: 'euclid'
      defined but not used [-Wunused-
      function]
Undefined symbols for architecture
      x86_64:
  "_euklid", referenced from:
      _main in ccncBu2o.o
ld: symbol(s) not found for
      architecture x86_64
collect2: error: ld returned 1 exit
      status
>
```

Du bist gewarnt!

```c
/* euclid-fail3.c -- faulty too */

// announce function to compiler
static int euclid(int, int);

int main()
{
   return euclid(45,30);
}

// defines the function
int euclid(int x, int y)
{
  if (x=y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```

```
> cc -std=c99 euclid.c -o euclid-fail3
>
> ./euclid-fail3
> echo $?
30 ← Uuups!
>
> gcc -std=c99 euclid.c -o euclid -Wall
euclid-fail3.c: In function 'euclid':
euclid-fail3.c:12: warning: suggest
parentheses around assignment used as
truth value
>
```


Aber ich dachte...

```c
/* euclid3.c -- calculates GCD with I/O */
#include<stdio.h>  /* printf */
#include<stdlib.h> /* atoi   */

static int euclid(int,int);

int main(const int c, const char* v[])
{
  printf("GCD is %d\n", euclid(atoi(v[1]),
        atoi(v[2])));
  return 0;
}

int euclid(int x, int y) {
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```

```
> cc -std=c99 euclid3.c -o euclid3
>
> ./euclid3
Segmentation fault
> ./euclid3 45 30
GCD is 15
```


Aber ich dachte... (Forts.)

- Der Fehler hier lag darin, dass eine **Annahme** über eine Laufzeitbedingung nicht gilt
- Kein Compiler kann ahnen, was zur Laufzeit passiert ➡ keine Warnung
- Abhilfe: Annahme explizit machen und **defensiv** programmieren:

```c
/* euclid4.c -- calculates GCD, defensive version  */
#include<stdio.h>  /* printf */
#include<stdlib.h> /* atoi   */
static int euclid(int,int);

int main(const int c, const char* v[])
{
  if (c!=3) {      /* 1st argument is program name */
    printf("Error: unsufficient number of arguments\n");
    return 1;
  }
  printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
}

int euclid(int x, int y) {
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```


Aber ich dachte... (Forts.)

- Da ständig Annahmen gemacht werden, gibt es eine generische Lösung: `assert`
- Um assert zu nutzen, muss die Datei „assert.h“ eingebunden werden: `#include<assert.h>`

```c
/* euclid5.c -- calculates GCD, using assert  */
#include<stdio.h>  /* printf */
#include<stdlib.h> /* atoi   */
#include<assert.h> /* assert */
static int euclid(int,int);

int main(const int c, const char* v[])
{
  assert(c==3);
  printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
}

int euclid(int x, int y) {
  if (x==y) return x;
  else if (x>y) return euclid(x-y,y);
  else return euclid(y-x,x);
}
```

Fehler

- Es gibt offensichtlich mindestens drei Sorten von Fehlern
  - Formal „falsches“ C ➡ **Syntaxfehler**[5] (Compilerfehler)
  - Fehlende Objekte (z.B. Funktionen) oder Namenskollision ➡ **Linkfehler**
  - Es wird etwas anderes gemacht, als beabsichtigt ➡ **Semantikfehler**[5](Logikfehler)

- Syntaxfehler und Linkfehler i.d.R.[6] werden zur Übersetzungszeit bemerkt
- Semantikfehler können zwei Ursachen haben
  - Der Algorithmus wurde nicht richtig umgesetzt
  - Der Algorithmus ist falsch oder nur unter bestimmten Bedingungen korrekt

Merke

Im ersten Fall (falsche Umsetzung) schöpft der Compiler evtl. einen „Verdacht“ und gibt (bei Anforderung) eine Warnung aus.
Es empfiehlt sich, jeder Warnung nachzugehen, u.U. sogar die Compileroption „`-Werror`“ zu nutzen.

[5] Die Begriffe „Syntax“ und „Semantik“ werden in späteren Kapiteln noch genauer besprochen.
[6] Es gibt pathologische Linkfehler, die erst zur Laufzeit auftreten.

2.6 Aufgaben

Aufgabe 2.1

Mit Hilfe des UNIX-Befehls „`time`“ kann man die Ausführungszeiten von Programmen bestimmen.
Machen Sie sich mit der Funktion von „`time`“ vertraut (`man time`) und vergleichen Sie die Ausführungszeit eines C-Programms mit einem äquivalenten Python-Programm!

Aufgabe 2.2

Nehmen Sie an, dass für den Euklidschen Algorithmus, wie er z.B. in „`euclid3.c`“ angegeben wurde, nur Parameter aus dem Bereich $[1, \ldots, 100]$ verwendet werden.
Wie häufig wird dann die Funktion „`euclid(int,int)`“ für eine ggT-Berechnung maximal aufgerufen?