gcc Ablauf für eine „hello.c“ Datei.

1. Pre-Prozessor (Zeilen im Quelltext mit # werden hier interpretiert): hello.c → hello.e
   GCC -E HELLO.C > HELLO.E
2. Compiler: hello.e → hello.o
   GCC -C HELLO.C
3. Linker (Bindet Objekt-Datei (xxx.o) mit Librarys zusammen): hello.o → a.out / hello
   GCC HELLO.C [-O HELLO]

Alle komplieren:
gcc *.c (dafür braucht es die Header Datei file.h, die alle Funktionsdeklarationen enthält [außer main]) ⇒ alle Dateien werden in eine kompiliert

Math:
GCC CODE.C **-LM**

```c
int main(int argc, char* argv[]){ ... }
// argc: #Parameter   argv[i]: Parameter (argv[0]: Programmname)
-----------------------------------------
char vBuf[128];
fgets(vBuf,128,stdin);
myInt=atoi(vBuf); // ganze Zahlen
myFloat=atof(vBuf); // Gleitkomma
-----------------------------------------
printf("%04d",i); // integer mit führenden Nullen
%d Dezimalwert
%p Adresswert (braucht &i)
%.5f Float mit 5 Nachkommastellen
%-30s String bzw. char-Array der linksbündig max bis zur 30. Stelle im Terminal ausgegeben wird
-----------------------------------------
int a=31,b=3;
a = a*10/b; // a = 310/3 = 103.33333
printf("%d.%d\n",a/10,a%10); // Ausgabe 3.3 (anstatt nur 3)
-----------------------------------------
int i=1;
double x = 5.0, y=1.0, summand = 1.0;
while (summand>0,00005){  // e^x=1+x/1!+x*x/2!+...
  summand = summand *x/i;
  y += summand;
  i++;
}
-----------------------------------------
while (operator != toupper('q')){
// prüft 1. Zeichen der Eingabe
  fgets(buf, 128, stdin);
  operator = buf[0]; }

while (1){
  fgets(buf, 128, stdin);
  if (!strncmp(buf, "qq", 2)) break; }
  // prüft 1. zwei Zeichen
-----------------------------------------
type myArr[] = {...};
for (i=0; i<sizeof(myArr)/sizeof(type); i++){
  myArr[i]=...
} // durch Array iterieren
```

Eingebaute Datentypen
Float: kann einige Zahlen (ganzzahlig) nicht darstellen (bspw. 2), hat Probleme sehr große und sehr kleine Zahlen miteinander zu addieren (durch Normierung der Exponenten kann die kleine zu 0 werden, oder Nachkommastellen verloren gehen)
Achtung: boolean kein Datentyp in C: ↷ Abfrage von true/false durch int: $int = 0 \mathrel{\widehat{=}} false \quad int \neq 0 \mathrel{\widehat{=}} true$

2er Komplement positive Zahl: $0110\,1100$
Negation: $1001\,0011 \;+1$
⇒Komplement: $1001\,0100 = -108 = 0x94$

Variable 4 Kennzeichen einer Variable:

- Adresse im Speicher (Ort)
- Datentyp (Verarbeitungsbreite)
- Bitkombination (Wert)
- Symbolischer Name

Ein Vektor fasst mehrere Variablen gleichen Datentyps unter einer zusammen.

Bei der Initialisierung hat die Variable einen Ausgangswert:

- Initialisierung innerhalb einer Funktion: zufälliger Wert (alte Speicherbelegung)
- Init. außerhalb einer Funktion: 0

```c
char c='c'; // 'c'=99 (ASCII)
char c=99;
int i=8, j=5, k;
char c=99, d='d';
float x=0.005, y=-0.01, z;
z=i/j;  // i/j wird in int gerechnet, also 8/5=1 und nicht 1.6! - z ist dann trotzdem float (Wert: 0.000...)
z=k=x;  // k=x=0 (wird abgeschnitten), also z=0.00000...
k=j=5?i:j;  // das selbe wie k=((j==5)?i:j); ist j=5? Wenn ja, dann k=i. Wenn nein, dann k=j.
printf("%d\n", i=!6 );  // !6 entspricht !(ungleich Null)=0
printf("%d\n", i&j ); // i bitweise mit j Verknüpft (ge-UND-et):
//  00001000
// &00000101
// =00000000
```

Simple Sort

```c
int data[] = {7,3,9,2,5};
int main(){
  int ige, iro;
  for (irt=0; irt<(5-1); irt++={
    for (ige = irt+1, ige<5, ige++){
      if (data[ige] < data[irt]){
        int tmp = data [ige];
        data[ige] = data[irt];
        data[irt] = tmp;
      }   }   }   }
```

Alphabetische Sortierung:
Jede Zeile bzw. jeden Array-Eintrag mit den folgenden vergleichen und vertauschen, wenn kleiner (jedes char durchgehen,

```c
#define N 10 // Länge der Zeichenkette
char data[][N] = {"Max", "Moritz", ...};

void printArr(char arr[][N]){
  int i,j;
  for(i = 0; i <  sizeof(data)/N; i++) {
    for(j = 0; j < N; j++) { printf("%c", arr[i][j]);  }
    printf("\n");
  } }

main(){
  int rowA, rowB, x;
  for (rowA = 0; rowA<(N-1); rowA++){
    for (rowB = rowA+1; rowB<N; rowB++){
      x = 0;
      for (x=0; data[rowA][x] == data[rowB][x] && data[rowA][x]!=0; x++){
        ;
      }
      if (data[rowA][x] > data[rowB][x]){
        char tmp;
        for ( ; x<N; x++){
          tmp = data[rowA][x];
          data[rowA][x] = data[rowB][x];
          data[rowB][x] = tmp;
        }   }   }   }
  printArr(data);
}
```

Ausdrücke Unäre Operatoren (bspw. − (negativ-Zeichen), ++ (Inkrementierung) oder Klammern(cast))
Binäre Operatoren (bspw. +, − (Rechenzeichen), <= usw.)

```c
int i;
long d;
i=(int)d; // cast: Typwandlung
i++; // Postfixoperator (wird im Rahmen eines groesseren Ausdrucks als letztes ausgefuehrt)
i=1;
j=6;
k=j+i++; // k=7, i=2
++i; // Praefixoperator (wird im Rahmen eines groesseren Ausdrucks als erstes ausgefuehrt)
i=1;
j=6;
k=j+ ++i; // k=8, i=2
```

Andere Zeichen: ∧ = XOR, ∼ = Bit-weise Negation, << = shift (nach links) (Bsp. i=4; i= i << 2; ⇒ i wird 16: 00000100 << 2 ⇒ 00010000)

```c
while (x<5){ ... }
-----------------------------------------
do{ ... } while (x<5);
```

Abbrechen der Schleife: break

```c
while (1){  ...
  if (x<5) break; }
```

Abbrechen der aktuelle Iteration (reset der Schleife): continue

```c
while (x<5){  ...
  if (x<4) continue;
  printf(...); }  // printf wird nur bei x>=4 ausgeführt
-----------------------------------------
for (i=1 ; x < 5 ; i++){ ... }
-----------------------------------------
switch (i){ // i ist ganzzahliger Ausdruck
  case 1: // wenn 1
    ... break;
  case 2 ... 5: // zwischen 2 und 5
    ... break;
  default:
```

```
    ...
}
```

Zeichenketten

```
fgets(buf, 128,stdin);
buf[strlen(buf)-1]=0; // an der Stelle strlen(buf) liegt
     die terminierende Null, an strlen(buf)-1 die
    return-Taste der Eingabe
puts(buf);  // puts gibt gesamten String aus, printf
    muss drüber iterieren
while (buf[i]!=0)
  printf("%c", buf[i++]);
```

Funktionen Wenn kein return_type gewählt wurde, dann default: INT.
Wenn kein return_type gebraucht wird, gibt man VOID an.
Speicherklassen:
AUTO (automatische Variable): wird vom Stack erzeugt (Kellerspeicher)
lokale Varibalen
EXTERN: Variable, die in einem anderen Kontext vereinbart ist
STATIC: leben bis zum Programmende, global-statische Variabeln werden nicht exportiert, immer initialiesiert, default 0
REGISTER: Variablen werden nach Möglichkeit in ein Prozessorregister gelegt (schnell)
VOLATILE: Variabeln werden immer im Hauptspeicher abgelegt (Gegenteil von register)

```
long fakult(int x); //Funktionsdeklaration (Prototyp)
```

Header-File

```
#include "fe.h" // wie bspw. stdio.h: eigene Datei in
    anderen QT
```

Pointer

```
*x = &i // x verweist auf die Adresse von i. Ausgabe von
     x gibt nur Adresse. *x (oder x[0]) (
    Dereferenzierung) ergibt Wert von i
```

Achtung: Bei Übergaben von Arrays (bspw. in Funktionen) wird nur Pointer auf das erste Element übergeben. Somit ist daraus auch nicht die Länge berechenbar. Des weiteren wird beim modifizieren der Daten im Array das original-Array überschrieben (da es nicht als Kopie, sondern als Verweis übergeben wurde)!

Berechnen Array Länge in Funktion (wenn nicht übergeben):

```
int mystrlen1(char *p){
  int i;
  for (i=0; p[i]!=0; i++);
  return i;
}
// oder genau so gültig (mit Pointer gerechnet):
int mystrlen2(char *p){
  int count;
  while (*p++)
    count++;
  return count;
}
```

Rekursion

- Rechtsrekursion (erst etwas rechnen, dann in die Rekursion gehen → fakultr)
- Linksrekursion (erst Rekursiv aufrufen, dann etwas ausführen → printu)

Eine Links- oder Rechtsrekursion lässt sich auch Iterativ darstellen.

- Zentralrekursion

Eine Zentralrekursion lässt sich nicht Iterativ darstellen.

Problem Rekursion: Es lässt sich nicht vorhersehen, wie viel Speicher benötigt wird. Da ist die Schleife leichter überschaubar.

Benutzerdefinierte Datentypen

Enum:

Aufzählungstyp (festgesetzte Bezeichnungen auf einen integer-Wert).

Struct:

Zusammenfassung von mehreren Komponenten (unterschiedliche eingebaute Dateitypen als un- intialisierte Variablen), die durch einen Namen beschrieben werden. Verwendung zur Modellierung eines Sachverhalts (wie im Beispiel Student mit seinen Eigenschaften).

Typedef:

Es wird ein synonymer Typname für einen existierenden Typnamen erstellt. So kann die Variableininitialisierung verkürzt werden (im Skript: struct tStudent→tStud).

Union:

Datensätze werden im Vergleich zum Struct übereinander geschrieben (Sinnvoll, wenn Unterstrukturen gleiche und auch ungleiche Eigenschaften haben. Beispielsweise Diplom- und Bachelor-Studenten).

Struct Struktur: alle Elemente liegen hintereinander (nicht zwangsläufig unmittelbar hintereinander) im Speicher

```
typedef int myint;
struct myStruc{
  char name[30];
  int Nummer;
}
typedef struct { ... } myStruc2;
sizeof(struct myStruc);
sizeof(myStruc2);
void putStr(struct myStruc s){
  puts(s.name);
}
void putStrP(myStruc *s){
  puts(s->name); // oder: puts( (*s).name );
}
struct myStru stru = {"String", 123};
```

enum

```
enum tWoTa{
  Montag=1, // bei Montag=1, fängt's bei 1 an zu zählen
      (anstatt 0)
  Dienstag,
  Mittwoch,
  Donnerstag,
  Freitag,
  Samstag=Freitag+1+0x10, // verändert das Wochenende
  Sonntag};
```

memory-allocation

```
int i;
tStud s;
int anz = 0;
tStud *ps = NULL, *psx;
while (weiter == 'y'){
  s = getStud();
  if(ps==NULL){ // Wenn noch kein Speicher freigegeben
    ps=malloc(sizeof(tStud));
    if (ps) {exit(-1);}
  } else {  // Sonst Speicher erweitern
    psx=realloc(ps,(anz+1)*sizeof(tStud));
    if (psx){
      ps=psx;
    } else { exit(-1);}
  }
  *(ps+anz) = s;  // Adresse vom freigegebenen Speicher
  anz++;
}
// am Ende Speicher wieder freigeben. Achtung: psx ist
    nur Zeiger auf ps
free (ps);
```

Verwendung von malloc/realloc:
Speicher nach Bedarf aus dem heap.
malloc hat als Ausgabe void* (generischer Pointer). Dieser ist nicht derefernzierbar und zuweisungskompatibel zu jedem getypten Pointer. Man kann mit ihm ebenfalls nicht rechnen (keine Arithmetik).
malloc: Speicher für Variable frei machen
realloc: freigemachten Speicher erweitern
free: Speicher wieder freigeben

Listen

Ringliste:

```
// Strukturtyp für Konnektor (Element mit Inhalt):
typedef struct TCNCT{
  struct TCNCT* next; // tCnct geht noch nicht innerhalb
      !
  void *pItem;  // void für generische Daten
}tCnct;
typedef struct{
  tCnct* pFirst;
  tCnct* pLast;
  tCnct* tCurr;
}tList;
```

Listenimplementation:

```
// erzeugt leere Liste:
tList *CreateList(void){
  tList* ptmp;
  ptmp=malloc(sizeof(tList));
  if(ptmp!=NULL){ // offene Liste: anfängliches tList
      hat nur NULL-Pointer
```

```
    ptmp->pFirst=ptmp->pLast=ptmp->pCurr=NULL;
  }
  return ptmp;
}
// hinten einfügen:
int InsertTail (tList* pList, void *pItemIns){
  // Verschieden Situationen: Anfügen an leere oder
      schon vorhandene Liste
  tCnct *ptmp = malloc(sizeof(tCnct));
  ptmp->next=NULL;
  if(ptmp){
    ptmp->pItem = pItemIns; // Connector mit Inhalt fü
        llen
    if (pList->pFirst!=NULL){ // Liste Leer
      pList->pFirst=pList->pLast = ptmp;
    } else {  // Liste enthält schon Konnektoren
      pList->pLast->next=ptmp;  // Das vorher letzte
          Element zeigt nun auf das eingefügte und damit
           neue letzte Element
      pList->pLast = ptmp;      // das neue letzte
          Element
    }
    pList->pCurr=ptmp;  // Das Element, mit dem zuletzt
        hantiert wurde ist pCurr
  }
  return (int)ptmp;
}
```

Unterschied: Offene Liste und Ringliste. Offene Liste startet mit NULL-Zeigern.

Oder: doppelt verkettete Ringliste. Vorteil: Jedes Element hat einen Vorgänger und einen Nachfolger. Dadurch reicht eine Funktion, die nach einem Element ein neues einfügen kann. Das kann an beliebiger Stelle passieren.

Bäume

```
typedef struct{
  void *pdata;
  struct TNODE* px[2];
} tNode
tNode treeInit={};
char* data[]={"moritz", "paul",  NULL);
tNode *pTree;
int mycmp(void*p1, void*p2){
  return (strcmp((char*)p1, (char*)p2>0)?0:1;
}
void addToTree(tNode *pt, void* pdata,  int (*fcmp)(void
    *, void*)){
  int i;
  if(pt->pdata == NULL){
    pt->pdata=pdata;
  } else {
    i=fcmp(pt->pdata, pdata);
      if(pt->px[i] == NULL){
        pt->px[i] = malloc(sizeof(tNode));
        *(pt->px[i]) = treeInit;
      }
      attToTree(pt->px[i],pdata, mycmp);
  }
}
  char *ptmp;
  char **p=data;
  // erstes Node erstellen (leer):
  pTree = malloc(sizeof(tNode));
  *pTree = treeInit;
  while  (*p){
    addToTree(pTree, *p, mycmp);
    p++;
  }
  while (1){
    fgets(buf,128,stdin);
    buf[strlen(buf)-1]=0;
    ptmp=malloc(strlen(buf)+1);
    strcpy(ptmp,buf);
    addToTree(pTree, ptmp, mycmp);
}
```

Dateiarbeit in C

```
FILE * pf;
pf=fopen("myFile.txt","rt"); // ("Dateiname inkl. Pfad",
    "Modus")
```

b/t	Texdatei ([t]) / Binärdatei(b)
r	Lesen öffen
w	Scheiben (überschreiben), ggf. erzeugen
a	Schreiben am Dateiende, ggf. erzeugen
r+	Lesen und Schreiben(ändern)
w+	Lesen und Schreiben(überschreiben), ggf. erzeugen
a+	Anfügen, Lesen, Erzeugen, ggf. erzeugen

Als Binärdatei speichern

```
...
FILE *pf;
...
  pf = fopen("Studs.bin","rb");
  if(pf){
    // Dateigröße ermitteln:
    fseek(pf,0,SEEK_END);
    anz=ftell(pf)/sizeof(tStud);
    rewind(pf);
    // Daten lesen
    for (i=0; i<anz; i++){
      if(ps==NULL){
        ps=malloc(sizeof(tStud));
        if (ps==NULL){puts("malloc hat nicht geklappt");
            exit(-1);}
        *ps=readStud(pf);
      } else {
        psx=realloc(ps,(anz+1)*sizeof(tStud));
        if (psx){
          ps=psx;
          *(ps+i)=readStud(pf);
        } else { puts("realloc hat nicht geklappt.");
            exit(-1);}
      }
    }
    fclose(pf);
  }
...
  // Daten speichern
  pf=fopen("Studs.bin","wb");
  if(pf==NULL){puts("fopen (write) hat nicht geklappt");
      exit(-1);
  for (i=0; i<anz; i++){
    writeStud(pf,ps+i);
  }
  fclose(pf);
...
tStud readStud(FILE* f){
  tStud s;
  fread(&s, sizeof(tStud),1,f);
  return s;
}
void write Stud(File* f, tStud* pStud){
  fwrite(pStud, sizeof(tStud),1,f);
}
```

Als Textdatei speichern (alternativ und ergänzend zum obigen Beispiel)

```
int getAnz(FILE* pf);
tStud readStud(FILE* pf);
void writeStud(FILE* pf, tStud* ps);
// Zeilen zählen und durch 4 teilen
int getAnz(FILE *pf){
  char buf[128];
  int n=0;
  while (fgets(buf, 128, pf)) n++;\\
  fseek(pf, 0, SEEK_SET);
  return n/4;
}
tStud readStud(FILE* pf){
  tStud s={};
  char buf[128];
  if(fgets(buf, 128, pf)){
    buf[strlen(buf)-1]=0;
    s.name = malloc(strlen(buf+1));
    if (s.name) strcpy(s.name,buf);
    else fprintf(stderr, "malloc faild in readStud\n");
    fscanf(pf,"%d\n%d\n%f\n",&s.matrNr,&s.belNote,&s.
        klNote);
  }
  return s;
}
void writeStud(FILE *pf, tStud* ps){
  fprintf(pf,"%s;%d;%d;%f\n",ps->name,ps->matrNr,ps->
      belNote,ps->klNote);
}
...
  // Einlesen aus Datei
  pf=fopen("Studs.txt", "rt");
  if (pf){
    anz=getAnz(pf);
...
  // Schreiben
  pf=fopen("Studs.txt", "wt");
```

Als CSV-Datei speichern Ähnlich wie bei .txt, bloß trennt man die Datensätze durch Semicolon und nicht durch neue Zeile.

Sinnvolle Funktion zur Zerlegung der Datensätze:

```
strtok(buf,";\n");
```

Funktionspointer

```
typedef void f(void);
f* pf;  // Das ist der Funktionspointer
```

oder:

```
typedef void (*tpf)(void);
tpf pf; // Das ist auch ein Funktionspointer
```

Anwendungsbeispiel:

```
void fxyz(void){
  puts("xyz");
}
pf = fxyz;
fxyz();
pf(); // ruft beides fxyz() auf!
```

Beispiel:

```
typedef void (*tpf)(int i);
tpf pFunc;
  pFunc=printDec;
  pFunc(x);
  pFunc=printHex;
  pFunc(x);
```

Preprozessor Zeilenverlängerung

```
int ma\
in(){
  puts{Spass\
  in c");
}
```

include

```
#include <...>  // für Systemheaderfiles (zu finden
    unter usr/include)
#include "..."  // für Applikationsheaderfiles (eigene)
```

Mehrfaches Einbinden u.ä. kann unterbunden werden durch:

```
#ifdef _H_DEBUG_
#define _H_DEBUG_
#include <stdio.h>
#endif
```

Symboldefinitionen

```
// Allgemein:
#define SYMBOL Tokensequenz
// Beispiel:
#define N 10
...
int inArray[N]; // Preprozessor ersetzt N mit 10
#define LEN 30 + 1
// KLAMMERN SETZTEN! ->
// #define LEN (30 + 1)
...
char name[LEN*3];
// Achtung: LEN wird vorm Ausrechnen ersetzt.
// Also: LEN*3 entspricht 30+1*3 und nicht (30+1)*3
```

Darauf ist zu achten:

- Tokensequenz (bei Zahlen) am besten Klammern.
- kein Semikolon!
- define-Konstruktion muss auf einer Zeile stehen.

Parameterbehaftete Macros

```
#define SYMBOL(<parameterlist>) Ersatztokenfolge
// Wichtig: Runde Klammer muss unmittelbar hinter SYMBOL
    stehen
#define SYMBOL(x) "str1" #x "str2"
// Bewirkt Verkettung der Strings mit dem Parameter
#define SYMBOL(x,y) x##y
// Ein neues Token ensteht im C-Quelltext aus x und y
```

Beispiel:

```
#define PYTHAGORAS(a,b) sqrt(a*a + b*b)
// Klammern wieder wichtig:
// #define PYTHAGORAS(a,b) sqrt((a)*(a) + (b)*(b))
#define STR1(x) "Max " #x " Moritz"
```

Darauf ist zu achten:

- Parameter im Macro klammern
- keine Seiteneffekte programmieren (Inkrementierung, Funktionsaufruf usw.)

Vordefinierten Symbole bspw.:

```
__FILE__ // Quelltext-Dateiname
__DATE__ // Datum, zu dem das Programm kompiliert wurde
```

Bedingte Übersetzung

```
#if <const-expr>
#if defined <symbol>
#ifdef <symbol>
#if !defined <symbol>
#ifndef <symbol>
#else
#elif <const-expr>
#endif
```

Bsp.:

```
#ifdef DEBUG
  printf("Debuginformation");
#endif
```

Gibt Debug-Information nur bei GCC PROGR.C -DDEBUG aus.

Funktionen mit variabler Argumentliste

vgl.: printf() mit beliebig vielen Argumenten abhängig von %d usw. im ersten String.

1. wenigstens ein fester Parameter
2. dann folgt, ...
   (Es können also weitere Parameter folgen. Typ und Anzahl der Parameter ist unbekannt.)

Macros zum Umgang mit variabler Argumentliste:

```
#include <stdarg.h> // Voraussetzung für variable
    Argumentlisten

va_start(ap, la);
// ap: Argument Pointer ( va_list ap; )
// la: Last Argument (letzter Parameter mit Typ und
    Namen vor ...)
printf("Name: %s, MatrNr: %d \n", Stud.name, Stud.MatrNr
    );

x=va_arg(ap, type);
// x: Zielvariable für den Wert des Arguments (muss vom
    Typ des tatsächlichen Parameters sein)
// ap wird um sizeof(type) erhöht
// dieser Wert wird x zugewiesen

va_end(ap);
```

Bsp.:

```
#include <stdarg.h>
#include <recout.h> // Andere my...-Funktionen in Arg.
    tgz
#define xprintf(x) myprintf x

void myprintf(const char* fmt, ...){ // fmt bspw. ("
    Programm: %s\n", argv[0])
  va_list ap;
  char    *p;
  char    *pstr;
  int     ival;

  va_start(ap,fmt);

  for ( p=fmt; // p auf Anfang des Formatsteuerstrings
        *p;
        p++){
    if (*p!='%'){
      myputc(*p);
    } else {
      switch(*++p){
        case 's':
          pstr=va_arg(ap,char*);
          myputs(pstr);
          break;
        case 'd':
          ival=va_arg(ap,int);
          myputd(ival);
          break;
        default:
          myputc(*p);
          break;
      }
    }
  }
}

int main(int argc, char* argv[]){
  int i;
  if (argc>1)
    i = myatoi(argv[1]);
  else
    i = -1;


  myprintf("Programm: %s\n int Value: %d\n", argv[0],
      argc);
  myprintf("i: %d, %x, DoubleVal: %f, Char: %c,
            Adresse argv[0]: %p",i,i,d, argv[0][0], argv
                [0]);

  xprintf( ("Test xprintf: %s\n", argv[0]) );
```