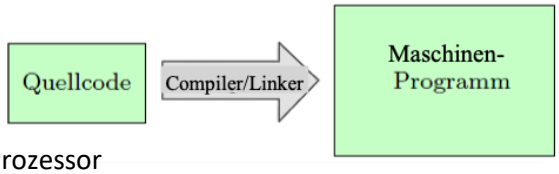
C: Kompilierung nach Maschinencode



- In C braucht man kein Laufzeitsystem
  - wie die JVM: Maschinencode
  - wird direkt vom Prozessor ausgeführt
- Übersetzte C-Programme sind spezifisch für einen bestimmten Prozessor

Deklaration und Definition

- Eine **Deklaration** informiert den Compiler darüber, dass es etwas gibt und wie man es benutzt.
- Eine **Definition** ermöglicht dem Compiler, Code oder Speicher für dieses Etwas zu erzeugen: Legt fest, wie dieses Etwas zu implementieren ist.
- Definition einer Variablen/Funktion darf nur 1x im gesamten Programm vorkommen
- *Deklaration: Angabe des Prototyps (nur Funktionskopf, ohne Rumpf)*
- *Definition: Angabe des Funktionskopfs mit Funktionsrumpf*

Headet Datei

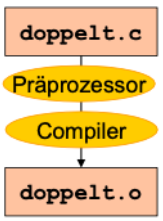
- **Header-Dateien enthalten nur Deklarationen** (z.B. Prototypen), keine Definitionen.
- **`#include`** kopiert den Inhalt der Header-Datei an die Stelle der `#include-Direktive`
- #ifndef (if not defined)
- #if #else • #elif • #endif

```
#include "int20.h"
#define LEN 20
extern struct int20 add20(int a, struct int20 b);
extern void test(void);

#ifndef S_H
#define S_H
struct int20 {
   char number[LEN];
};
#endif
```

Kompilieren

- Jede Quelldatei *.c wird kompiliert in Objektcode-Datei *.o
- .o Datei enthält den Maschienencode
  1. Präprozessor ersetzt **`#...`** – Anweisungen (rein textuell)
  2. Der eigentliche Compiler übersetzt die Ausgabe des Präprozessors in Objektcode (und prüft auf Korrektzeit und meldet ggf. Fehler) und speichert den in Objektcode- Datei (*.o-Datei)



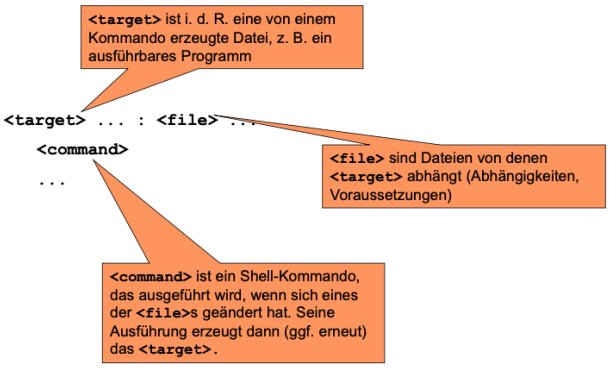
Linken

- Objektcode-Datei enthält zwar schon Maschinencode, ist aber für sich allein noch nicht ausführbar
- Der Linker bindet alle benötigten Objektcode-Dateien zusammen ("linken") zu einem ausführbares Maschinenprogramm

Makefiles

- Überprüft Änderungszeiten und Abhängigkeiten von Dateien.
- Ein Makefile besteht aus Regeln.
- Die Regeln beschreiben die Abhängigkeiten zwischen Dateien.

***Erinnerung**: Nicht nur dieses <target> , sondern auch alle dadurch jetzt veraltenen weiteren Targets werden neu erzeugt (und rekursiv so weiter)*



Makefile – Pattern rules

Vordefinierte Variablen für Pattern-Rules:

- **`$<`** - Erste Datei in der Liste von Voraussetzungen
- **`$^`** - Alle Voraussetzungen
- **`$@`** - Name des Targets

```
GCC_ARGS = -std=c99 -Wall -pedantic-errors
OBJ = math.o summe.o differenz.o input.o

math: $(OBJ)
   gcc $(GCC_ARGS) -o $@ $(OBJ)

%.o: %.c
   gcc $(GCC_ARGS) -c $<

math.o: summe.h differenz.h
summe.o: input.h
differenz.o: input.h

clean:
```

Phony targets – **`clean`**

Kontrollstrukturen

Bedingte Anweisungen

- *if (…) { … } else { … }*
- *switch (…) { case … }*

Schleifen

- *for (…; …; …) { … }*
- *while (…) { … }*
- *do { … } while (…)*

Datenorganisation in C

- In C sind die Speichergrößen und Wertebereiche nicht einheitlich festgelegt.
- `sizeof([..])`:

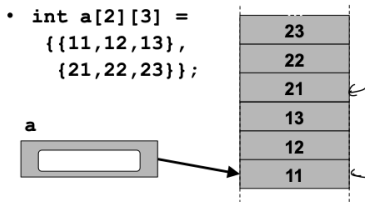
```
char = 1;
short = 2;
int = 4;
long = 8;
float = 4;
double = 8;
```

prinftf([…]);

```
printf("%d\n", i); // int in Dezimalschreibweise (%d) ausgeben
printf("%c\n", c); // char ausgeben
printf("%f\n", d); // double ausgeben (%f, da %d schon vergeben)
printf("%s\n", s); // Ausgabe endet beim ersten \0-Zeichen:
printf("%p\n", a); // Ausgabe von Zeigern
```

Array-Benutzung

- C: Array ist einfach eine Folge von Elementen hintereinander im Speicher
- Wert der Konstante: **Speicheradresse des ersten Elements des Arrays**
- Länge des Arrays muss zur Compilierzeit feststehen!
- Bsp. `int a[] = {0, 2, 4, 6, 8};`



Mehrdimensionale Arrays

- C: Elementfolge im Speicher Arrayinhalt wird linearisiert.
- Alle Zeilen müssen gleich lang sein.
- **Für 2. - n. Dimension muss im Parametertyp die Größe angegeben werden**
- `printBlock(int a[][10][20])`

Zeichenketten ("Strings")

- In C gibt es keinen Typ für Zeichenketten. Stattdessen: char-Array benutzen
- Null Byte ans Ende des Char-Arrays „`\0`“

Funktionen in <string.h>

- `strcpy(s_ziel, s_quelle)` kopiert eine Zeichenkette. (ZK)
- `strcat(s_ziel, s_quelle)` hängt die ZK s_quelle an den Inhalt der Zeichenkettenvariablen s_ziel.
- `strlen(s)` liefert Länge einer Zeichenkette: Anzahl Zeichen ohne abschließendes \0.
- `strcmp(s1,s2)` vergleicht s1 und s2 und liefert … 0, wenn s1 und s2 gleich sind

Eingabe-Probleme durch Eingabepufferung

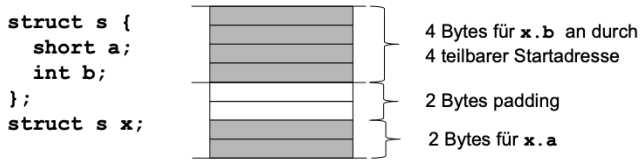
- Nach dem Einlesen empfiehlt es sich, weitere, nicht mehr benötigte Zeichen einzulesen und zu verwerfen, damit nachfolgende scanf-Aufrufe nicht durch sie gestört werden.
- ```
  scanf("%d", &alter);
  while (getchar() != '\n') ;
  scanf("%40[^\n]", name);
  ```

```
#ifndef S_H
#define S_H
struct angestellte {
   char name[NAME_LEN+1];
   int personalnummer;
   float gehalt;
};
#endif

struct angestellte schmitz;
strcpy(schmitz.name, "Schmitz");
schmitz.personalnummer = 1234;
schmitz.gehalt = 2752.44
struct angestellte weber = {"Schmitz", 1234, 2752.44};
```

Struktur-Typen

Speicher-Layout eines struct



- Die Elemente einer struct-Variablen liegen in der Reihenfolge in der Definition im Speicher
- C erlaubt keine Mehrfachdeklaration desselben structs

Benutzerdefinierte Typnamen

- typedef ermöglicht die Deklaration von Typen mit selbst gewählten Namen

Bsp.

- `typedef int kundennummer;` `kundennummer meineKnr = 4711;`
- `typedef struct angestellte angestellter;` `angestellter schmitz, mueller;`
```

Zeiger

- **Zeiger** = Adresse einer Speicherzelle (= Nummer der ersten Speicherzelle (1. Byte))
- Zugriff auf die referenzierte Variable (**"Dereferenzierung")** erfolgt indirekt:
    1. Auslesen der Adresse der referenzierten Variable aus der Zeigervariablen
    2. Auslesen des Wertes an dieser Adresse (Inhalt der referenzierten Variabl.)\

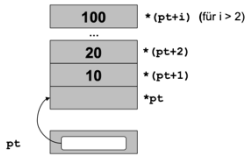
- **Zeigervariable sind typisiert**: Typ gibt u.a. an, wieviele Bytes zur referenzierten Variable gehören

Bsp.

`int i;`	/* int-Variable i deklarieren */	*Typ-Angabe*
`int* ipt;`	/* Zeiger-Variable ipt deklarieren */	*Typ-Angabe*
`ipt = &i;`	/* Adresse der Variable i */	*Zu einer Variable ihre Adresse finden ("Referenzieren"):*
`*ipt = 1;`	/* belegt i mit 1 (äquivalent: i = 1;) */	*Zu einer Adresse den Wert an dieser Adresse finden*
`(*ipt)++;`	/* erhöht die Variable i um 1 (äquivalent: i++;) */	*("Dereferenzieren"):*

Adressarithmetik

- Ist `pt` eine Zeigervariable, so ist **`pt+1`** die Adresse der nächsten Variabler im Speicher.
- **Zu einem Zeiger addierte Zahlenwerte bedeuten nicht Bytes, sondern Anzahl der übersprungenen Variablen im Speicher.**
- `*(pt+n) = ...;` (ist die Variable n*sizeof(T) Bytes "hinter" *pt)



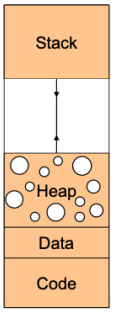

```
void cpyarr(int a[], int b[]) {
   for(int i = 0; i < arrlenght; ++i){
      *(b+(arrlenght-i-1)) = *(a+i);
     //b[arrlenght-i-1] = a[i];
   }
```

```
struct beispiel {
  short i;
  int j;
  char s[10];
};
```

```
void fill(char u[]){
  *(short *)u = 89;
  *(int *)(u+4) = 32168;
  strcpy(u+9,"Rosi");
}
```

Dynamische Speicherverwaltung

- Speicherplatz für **lokale Variablen** von C-Funktionen liegt auf dem Stack
- *Objekte selbst werden in Java immer auf dem Heap angelegt: Lebensdauer unabhängig von Methodenaufrufen*
- Ein Aufruf von **`malloc(size)`** allokiert einen Block von **`size`** Bytes auf dem Heap und gibt die Adresse des (ersten Bytes des) Blocks zurück



Malloc

- malloc liefert einen ungetypten Zeiger **`(void*)`**

```
typedef struct { … } ding ();
ding* d;
…
d = (ding*)malloc(sizeof(ding));
```

free

- mit malloc()belegte Speicherblöcke wieder explizit freigeben: Durch Aufruf von free(zeiger).

Empfehlung: Nach free() den Zeiger unbrauchbar machen durch Überschreiben mit NULL:

```
d = (…)malloc(sizeof(…));
…
free(d);

free(zeiger);
zeiger = NULL;
```

Zeiger auf Strukturen

```
array = (angestellter*)malloc(anzahl*sizeof(angestellter));
for (i = 0; i < anzahl; i++) {
   scanf("%s %d %f",
   array[i].name, &array[i].personalnummer, &array[i].gehalt);
}
```

Alternative: Speichere im Array nur Zeiger auf Angestellte:

```
angestellter* array[arraygröße] = { NULL };
array[i] = (angestellter*)malloc(sizeof(angestellter));
(*array[i]).personalnummer = 1234;
pt->personalnummer = 1234; // Kürzer
```

Zirkuläre Typdeklarationen

```
struct kante; /* Vorwärtsdeklaration */

struct knoten {
   int nummer;
   struct kante *kanten;
};
struct kante {
   int gewicht;
   struct knoten *start, *ende;
}
```

Zeiger auf Zeiger und Konstanten

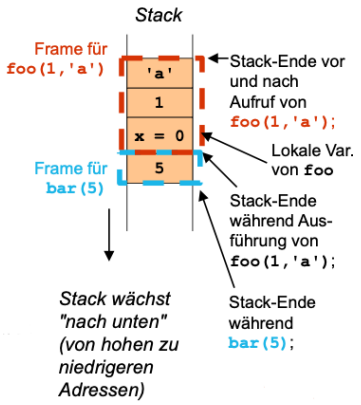
	`int i = 42;` `int* p = &i;`	Referenzierte Var. ändern `*p = 43;`	Zeiger „umbiegen“ `++p;`
1	`const int * p = &i;`	Nicht zulässig	Zulässig
3	`int * const p = &i;`	Zulässig	Nicht zulässig
4	`const int * const p = &i;`	Nicht zulässig	Nicht zulässig

Parameter und Rückgabewert

- Jeder Funktionsname darf nur einmal im gesamten Programm(!) verwendet werden
  → Fehler beim linken

Parameter-Übergabe über den Stack

- Parameter-Werte ("Argumente") werden von Aufruf-Code auf den Stack "geschoben" und vom Code der aufgerufenen Funktion von dort gelesen
- Für jeden Funktionsaufruf (nicht nur für jede Funktion!) gibt es einen Bereich auf dem Stack, wo die Argumente für diesen Aufruf gespeichert sind: **"Stack-Frame"**
- Aufgerufene Funktion legt ihre lokale Variablen auf dem Stack an (unterer Teil des Stackframes für diesen Aufruf)
- Bei Rückkehr aus der Funktion löscht zunächst der Aufgerufene, dann der Aufrufer den seinen Teil wieder



Call by value

- Parameter werden in C immer als Wertkopie an Funktionen übergeben:
  Nicht die Variable selbst, sondern ihr aktueller Wert wird übergeben **("call by value")**

Call by reference

```
void tausch(int* a, int* b) {
    int hilf;
    hilf = *b; *b = *a; *a = hilf;
}
```

"Referenz" auf `int`-Variablen übergeben in Form von `int*`
"Call by reference"

Array als Rückgabewert ist nicht erlaubt. → Stattdessen benutzt man … **Zeiger als Rückgabewert**

- Falls man ein in der Funktion (per mallloc) erzeugtes Array zurückgeben will, gibt man einen Zeiger auf sein erstes Element zurück.

Bsp

```
struct angestellter* maxGehalt(struct angestellter* a,
struct angestellter* b) {
if (a->gehalt > b->gehalt) return a;
else return b; }
```

Rückgabewert der Funktion main

- 0 alles in Ordnung
- !0 signalisiert Fehler!

Parameter der Funktion main

Wie in Java ein Array von Zeichenketten als Parameter möglich:
main(int argc, char* argv[])

- argc = „argument count“ (Anzahl)
- argv = „argument vector“
- argv[0] enthält den Namen der Datei mit dem (ausführbaren) C-Programm.
- argv[1] bis argv[argc - 1] enthalten die eigentlichen Parameterwerte.
- argv[argc] enthält den Nullzeiger NULL

Funktionen und Funktionsdeklarationen

- C kennt kein Überladen von Funktionen

Speicherklassen

- Lokale Variable: **`auto`**
  - Lebensdauer: angelegt - Verlassen des Blocks
- Globale Variable: **Variable außerhalb von Funktion/Block definiert ("file scope")**
  - Sichtbarkeit: Gesamtes Programm
- Modul-lokale Variable: **Variable außerhalb von Funktion definiert, aber mit Schlüsselwort static in Definition**
  - Sichtbarkeit: Nur in dieser Quelldatei ("Modul") (aber erst ab Definitionsstelle!)
- **`static`** für lokale Variable
  - Sichtbarkeit: Wie normale lokale Variable (d.h. außerhalb des Blocks unsichtbar) aber im Data-Segment abgelegt

Zeiger auf Funktionen

C-Syntax dafür: `int (*verarbFunc) (int i)`

- Name der Zeigervariablen: `verarbFunc`
- Der Rest definiert Rückgabetyp und Parameterliste der Funktion.

Allgemeines Format der Deklaration einer Funktionszeiger-Variablen:
`Rückgabetyp (*varname) (Parameterliste);`

Ein Array von Funktionszeigern deklarieren Sie wie folgt:
`Rückgabetyp (*varname[Arraygröße])(Parameterliste);`
Der Aufruf eines Elements des Arrays erfolgt so:
`Ergebnis= (*varname[Index])(Argumente);`

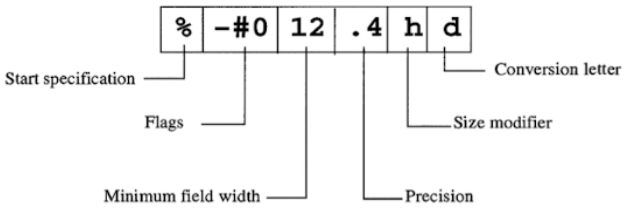
```c
void mwstNetto(double x) {                    //0=Mwst. vom Netto
    printf("Mwst. vom Netto: %.2f\n\n", (x*0.19));
}
void mwstBrutto(double x) {                   //1=Brutto vom Netto
    printf("Brutto vom Netto: %.2f\n\n", (x*1.19));
}
void Netto2Brutto(double x) {                 //2=Netto vom Brutto
    printf("Netto vom Brutto: %.2f\n\n", (x/1.19));
}

int main(void){
    int fnk = -1;
    double x =0;
    void (*funktionen[3])(double) = {mwstNetto,mwstBrutto,Netto2Brutto};

    do{
        output();
        scanf("%d %lf",&fnk, &x);
```

Printf

- Minimale Feldbreite (optional)
- Punkt (.) und Genauigkeitsangabe (engl. precision) (optional)



scanf: Formatiertes Lesen von Standardeingabe

- Rückgabe: Anzahl erfolgreich eingelesener Werte oder EOF (falls Dateiende).
- Maximale Feldbreite (optional)

Dynamische Datenstrukturen in C
Einfach verkettete Liste in C

```c
struct knoten {
    int wert; // … oder komplexere Daten
    struct knoten* next;
};

knoten* suchen(knoten* kopf, int gesuchter_wert) {
    knoten* laufzeiger;
    laufzeiger = kopf;
    while (laufzeiger != NULL && laufzeiger->wert != gesuchter_wert) {
        laufzeiger = laufzeiger->next;
    }
    return laufzeiger;
}

int einfuegen_kopf(knoten** kopfref, knoten* einzufueg) {
    if (einzufueg == NULL || kopfref  == NULL) return -1;
    einzufueg->next = *kopfref;
    *kopfref = einzufueg;
    return 0;
}
```

```c
int einfuegen_ende(knoten** kopfref, knoten* einzufueg){
    knoten* sucheAltesEnde;
    if (einzufueg == NULL || kopfref==NULL) return -1;
    if (*kopfref == NULL) {
        *kopfref = einzufueg;
        einzufueg->next = NULL;
    } else {
        sucheAltesEnde = *kopfref;
        while (sucheAltesEnde->next != NULL) {
            sucheAltesEnde = sucheAltesEnde->next;
        }
        einfuegen_nach(sucheAltesEnde,einzufueg);
    }
    return 0;
}
```

L6: Random-Zahl

```c
#include <stdlib.h>
int randomNumber(int hi){
    const double scale = rand()/((double)RAND_MAX+1.0);
    int i = (int)(scale*hi);
    return (i >= hi ? hi - 1 : i);
}
```

String -> Int

```
int atoi(const char* string)
```

```c
int checkError(char *endptr, char *input) {
    if (strlen(endptr) != 0) {
        printf("Kann '%s' nicht in Zahl umwandeln: Falsches Format\n", input);
        return 1;
    }
    else if (errno != 0) {
        printf("Kann '%s' nicht in Zahl umwandeln: %s\n", input, strerror(errno));
        return 1;
    }
    return 0;
}

int main(int argc, char *argv[]){
    long a1,a2;
    char* endptr = NULL;
    errno = 0;

    if (argc != 3) {
        printf("Benutzung: %s <zahl> <zahl>", argv[0]);
        return 1;
    }

    a1 = strtol(argv[1], &endptr, 10);        /* 10 = base = Dezimalsystem */
    if (checkError(endptr, argv[1]) == 1) return 1;
    return 0;
}
```

L7: Files lesen und schreiben

```c
void binaer_speichern(angestellter arr[],int anz){
    FILE* fp;
    fp = fopen(FILE_NAME,"wb");
    fwrite(&anz, sizeof(anz), 1, fp);
// (adresse zum lesen,block-groesse,anzahl,file)
    fwrite(&arr[0], sizeof(angestellter), anz, fp);
    fclose(fp);
}
```

```c
void binaer_laden_und_ausgeben(void){
    int n;
    FILE* fp;
    fp = fopen(FILE_NAME,"rb");
    fread(&n, sizeof(int), 1, fp);
    angestellter* ang = (angestellter*)malloc(n*sizeof(angestellter));
    fread(&ang[0], sizeof(angestellter), n, fp);
    fclose(fp);
}
```