C

Basierend auf den CDT1-Unterlagen des CDT Teams

Zusammengefasst durch Simon Flüeli

Autor	Simon Flüeli
E-Mail	fluelsim@students.zhaw.ch
Datum	22.05.2011
Fach	C und Digitaltechnik (CDT1)
Originalunterlagen	https://olat.zhaw.ch/olat/auth/1%3A-1%3A0%3A0%3A0/kube2010/kurs.view.jsp

Inhaltsverzeichnis

1 Grundlagen

1.1 Warum C

- Sehr effizienter Code
- Grosse Kontrolle bei Programmierer
- Effizienz bei Memory-Footprint und Ausführungsgeschwindigkeit
- Fördert Verständnis für unterliegendes System
- Viele verfügbare Bibliotheken

1.2 C und Java

- Java hat bewusst auf C aufgebaut
- Java hat einige Details weggelassen
- Keine automatische Garbage-Collection in C
- Pointer in C
- Variablen in C können wahlweise auf Stack oder auf Heap alloziert werden (in Java nur eine Möglichkeit)
- Keine Überprüfung von Array-Grenzen in C
- Java-Compiler generiert Bytecode
- C-Compiler generiert Maschinencode (nicht plattformunabhängig)

2 Grundlegende Programmelemente

2.1 Hello World

```
/* helloworld.c */
#include <stdio.h>

/* Hauptprogramm */
int main(void) {
      printf("Hello World in C\n");
}
```

2.2 Überblick über C

- Prozedurale Programmiersprache (!= objektorientiert)
  - keine Klassen
  - Funktionen statt Methoden
- Jedes C-Programm hat eine main-Funktion (Einstiegspunkt)

2.2.1 Schlüsselwörter in C

auto	break	case	char	continue	const	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	while	volatile			

2.2.2 Variablen- und Funktionsnamen

- Erlaubte Zeichen: 0…9; a…z; A…Z, und das Zeichen _
- Unterscheidung zwischen Gross- und Kleinschreibung
- Erstes Zeichen muss Buchstabe oder _ sein
- Keine Schlüsselwörter

2.2.3 Einfache (primitive) Datentypen

- C kennt nur vier elementare Datentypen

Datentyp	**Länge**	**Bereich**
char	1 byte	-128 bis 127
int	4 bytes	-2^{31} bis $2^{31}-1$
float	4 bytes	$-3.4*10^{38}$ bis $3.4*10^{38}$
double	8 bytes	$-1.79*10^{308}$ bis $1.79*10^{308}$

- Zusätzlich gibt es Qualifier (Attribute)

Datentyp	Länge	Bereich
unsigned char	1 byte	0 bis 255
unsigned int	4 bytes	0 bis $2^{32}-1$
short [int]	2 bytes	-32768 bis 32767
unsigned short [int]	2 bytes	0 bis 65535
long [int]	8 bytes	-2^{63} bis $2^{63}-1$
unsigned long [int]	8 bytes	0 bis $2^{64}-1$
long double	12 bytes	$-1.2*10^{4932}$ bis $1.2*10^{4932}$

- Werte sind hardwareabhängig, die obigen Werte sind typisch für heute übliche 32-Bit Architektur

2.3 Literale

- Im Code eingefügte, unveränderliche Werte

2.3.1 Ganze Zahlen (short, int, long)

- Zahlen, die mit 1…9 beginnen, werden dezimal interpretiert; z.B. 3987
- Zahlen, die mit 0 (Zahl Null) beginnen, werden oktal interpretiert; z.B. 037
- Zahlen, die mit 0X oder 0x beginnen, werden hexadezimal interpretiert; z.B. 0x23
- Default: Dezimalzahl wird als int interpretiert, oktale/hexadezimale Zahl als unsigned int
- Ist Zahl für int zu gross, wird sie als long interpretiert
- Soll Zahl explizit als long interpretiert werden, so wird l oder L angehängt
  35000l (oder 35000L)
  0104270l
  0x8868l

2.3.2 Gleitkommazahlen (float, double)

- Weisen Dezimalpunkt (z.B. 3.1415) oder Exponenten (z.B. 1e-2) auf
- Default: double, kann mit f oder F explizit als float bezeichnet werden
  3.1415f (oder 3.1415F)

2.3.3 Einzelne Zeichen (char)

- Werden durch '' eingeschlossen; z.B. 'A'
- Gespeichert wird ASCII-Wert (1 Byte) des Zeichens (A = 65)
- Spezielle Zeichen mittels \ (z.B. \n, \t, \\, \', \", \0)
- Einzelnes Zeichen kann auch durch ASCII-Wert (dezimal, oktal oder hexadezimal) ausgedrückt werden (65, \101, \x41 entsprechen alle dem ASCII Zeichen mit dem Wert 65 (= A))

2.3.4 Strings

- Durch Anführungszeichen eingeschlossene Zeichenfolge; z.B. "ZHAW"
- Es existiert kein Datentyp String -> Array von Characters (char)
- Strings werden automatisch durch ASCII Zeichen NUL abgeschlossen
  - "ZHAW" entspricht den fünf ASCII Zeichen 'Z' 'H' 'A' 'W' '\0'
  - Speicherplatz in Bytes: Anzahl Zeichen + 1

2.3.5 Symbolische Konstanten

- Werden mit #define am Anfang des Programms gesetzt
  #define MAX_LENGTH 1000
- Während Kompilieren wird jedes MAX_LENGTH im Code mit entsprechendem Wert (1000) ersetzt

2.4 Deklarationen

2.4.1 Variablendeklaration

- Deine Defaultwerte, ausser bei statischen Variablen

```
double hoehe;
int laenge, breite;
double r, radius = 15.0;
```

2.4.2 Konstantendeklaration

- Wert von pi kann einmal initialisiert, dann aber nicht mehr verändert werden

```
const double pi = 3.14159;
```

2.4.3 Typdeklaration

```
typedef int Index;
```

- Erlaubt Definition von neuen Namen für Datentypen; hier ist z.B. Index ein anderer Name (Synonym) für int, deshalb sind die folgenden Variablendeklarationen äquivalent

```
int i;
Index i;
```

2.5 Operatoren

2.5.1 Arithmetische Operatoren

+ - * / % (modulo)

2.5.2 Relationale Operatoren

Arithemtische Operatoren	+ - * / * (modulo)
Relationale Operatoren	> >= < <=
Logische Operatoren	&& ||
Gleichheitsoperatoren	== !=
Negationsoperator	!
Increment / Decrementoperator	++ --
Bitoperatoren	& (AND) | (OR) ^ (XOR) << (bit shift left) >> (bit shift right) ~ (Einerkomlement)
Zuweisung	= += -= *= /= %= &= ^= |= <<= >>=
Conditional Expression	? :
Referenzoperator, Dereferenzoperator	& *

- Sind beide Operanden bei der Division int-Werte, so wird Resultat auf ganzzahligen Wert abgerundet.
  5/3 = 1.667 -> wird zu 1

2.5.3 Order Precedence and Associativity

Symbol	Type of Operation	Associativity
`() [] . ->`	Expression	Left to right
`! ~ ++ -- + - * & (type) sizeof`	Unary	Right to left
`* / %`	Multiplicative	Left to right
`+ -`	Additive	Left to right
`<< >>`	Bitwise shift	Left to right
`< <= > >=`	Relational	Left to right
`== !=`	Equality	Left to right
`&`	Bitwise-AND	Left to right
`^`	Bitwise-exclusive-OR	Left to right
`|`	Bitwise-inclusive-OR	Left to right
`&&`	Logical-AND	Left to right
`||`	Logical-OR	Left to right
`? :`	Conditional-expression	Right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	Simple and compound ass.	Right to left
`,`	Sequential evaluation	Left to right

2.6 Ausdrücke

- Verbindung von Operanden mit Operatoren
  z.B. 1 + 3, u = 7 * r etc.
- Alle numerischen Typen werden implizit in grössten Typ gewandelt
  3 / 2 -> int / int -> grösster Typ = int -> Resultat = 1 (Typ int)
  3.0 / 2 -> double / int -> grösster Typ = double -> Resultat = 1.5 (Typ double)

2.7 Type Casts

- Explizite Konvertierung von einem Typ in einen anderen
- Gründe
  - Wandeln einer Gleitkommazahl in einen Integer Wert und umgekehrt
  - Wandeln eines void-Pointers in einen Pointer auf einen bestimmten Datentyp
  - Wandeln eines Zeigers auf eine Instanz der Basisklasse in einen Zeiger auf eine abgeleitete Klasse

-var = (type) expression

double d; int i = 5; d = i / 3; -> d = 1.0

double d; int i = 5; d = **(double) i** / 3; -> d = 1.6667

2.8 Kontrollstrukturen

- Können verschachtelt werden
- Einfache Anweisungen werden mit einem Semikolon abgeschlossen

```
c = a + b;
printf("Hello World\n");
```

- Block: Zusammenfassung von Anweisungen mittels geschweiften Klammern
- Variablen müssen in C am Anfang eines Blocks deklariert werden

```
{
   int a = 5; b, temp;
   b = a;
   temp = b;
}
```

2.8.1 If-Else / Else-If

- Ermöglicht Verzweigungen
- If-Else auch folgendermassen möglich: result = ((booleanExpression) ? value1 : value2)

```
result = -1;
if(n >= 0)  {
   result = 1;
}
```

```
if(n >= 0) {
   result = 1;
} else {
   result = -1;
}
```

```
if(n > 0) {
   result = 1;
} else if(n == 0) {
   result = 0;
} else {
   result = -1;
}
```

2.8.2 For-Schleife

- Ermöglicht das wiederholte Durchlaufen eines Programmabschnitts

```
int sum = 0;
int max = 5;
int i;

for(i = 1; i <= max; i++) {
   sum += i;
}

Das geht nicht in C
for(int i = 1; i <= max; i++) {
   sum += i;
}
```

2.8.3 While- und Do-While-Schleifen

- Ermöglicht das wiederholte Durchlaufen eines Programmabschnitts

```
int sum = 0;
int max = 5;
int i = 1;

while(i <= max) {
   sum += i;
   i++;
}

oder:

do {
   sum += i;
   i++;
} while(i <= max);
```

- Do-While Schleife wird immer mindestens einmal durchlaufen, da der Test erst nach einem Durchlauf geschieht.

2.8.4 Switch-Anweisung

- Ermöglicht individuelles Reagieren auf verschiedene Werte einer Variable

```
switch(n) {
   case 1: result = 1;
   break;
   case 2:
   case 3:
   case 4: result = 10
   break;
   default: result = 0;
   break;
}
```

- break ist notwendig, da sonst weitere cases abgearbeitet werden

2.8.5 Wo ist der Datentyp boolean?

- Es gibt keinen Datentyp boolean in C
- Regel: ein Ausdruck gilt als false, wenn er = 0 ist; sonst gilt er als true

```
int x = 10;
while(x > 0) {
   x--;
}

int x = 10;
while(x) {
   x--;
}
```

- Ausdruck muss kein ganzzahliges Resultat produzieren
  - while(1) {…}, while(1.5){…}, while(-3.1415){…] etc. in C entsprechen alle while(true){…} in Java

2.9 Input/Output

- Standard Library enthält unter anderem Input/Output Funktionen
- Werden mit #include <stdio.h> am Anfang eines Programms eingebunden
- Funktionen für einfachen Input/Output

puts("Hello");	Ausgabe des Strings Hello auf Standard Output
putchar('A');	Ausgabe des Zeichens A auf Standard Output
c = getchar(void);	Einlesen eines Zeichens von Standard Input

2.9.1 Funktionen für formatierte Ausgabe

- Generell: printf(format-string, arg1, arg2, …);

printf("Hello World\n");	Ohne weitere Argumente
printf("The sum of %d and %d is %d\n", a, b, a+b)	printf mit drei Argumenten, wobei a+b innerhalb der Funktion ausgewertet wird; %d bedeutet, dass a, b und a+b als int interpretiert werden

- %d nennt man Konvertierungsoperationen

2.9.2 Funktionen für formatierte Eingabe

- Generell: scanf(format-string, arg1, arg2, …);
- Beispiel : Einlesen von 3 Werte in die 3 int Variablen day, month und year

```
scanf("%d%d%d", &day, &month, &year);
```

- Achtung: der Adressoperator & ist wichtig
- Erst durch Eingabe des letzten Return werden die Zeichen aus dem Eingabebuffer gelesen
- Auch das letzte Return-Zeichen wird in den Buffer eingelesen und steckt nach Zurückkehren der scanf-Funktion noch im Buffer -> gegebenenfalls mit einzelnem getchar() auslesen
- Wird mit scanf ein double eingelesen, muss der Konvertierungsoperator %lf (statt %f) verwendet werden (bei printf funktioniert %f auch für double)

2.9.3 Konvertierungsoperatoren

- %d, %i (int); %u (unsigned int)
- %c (char)
- %s (char *, Zeichen des Strings werden ausgegeben bis \0 gefunden wird)
- %f (double, float)

- Bei Zahlen können zwischen dem % und dem Konvertierungszeichen die minimale Länge m (Anzahl Zeichen) des Outputs und die Anzahl Dezimalstellen d angegeben werden: %m.df

```
double d = 5.12345;
printf("%f", d);        /* Output: 5.123450 (default d = 6) */
printf("%.3f"; d);      /* Output: 5.123 */
printf("%10.3f", d);    /* Output:          5.123 */
```

3 Komplexe Datentypen

3.1 Aufzählungstyp

- Erlaubt die Definition einer Liste von konstanten int-Werten

```
enum wochentage {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
Sonntag};
```

- Dies weist den Tagen die Werte Montag = 0 … Sonntag = 6 zu
- Werte können auch explizit gesetzt werden

```
enum frucht {Apfel = 5, Birne = 10, Zitrone = 15};
enum frucht {Apfel = 5, Birne, Zitrone}; /* Birne = 6, Zitrone = 7 */
enum frucht {Apfel, Birne = -1, Zitrone} /* Apfel = 0, Birne = -1, Zitrone
= 0; Werte müssen nicht unterschiedlich sein! */
```

- Die konstanten Werte können in Ausdrücken verwendet werden
enum frucht {Apfel, Birne, Zitrone};

```
int essen = Apfel;            /* essen = 0 */
essen = Birne + Zitrone;            /* essen = 3 */
```

- Es können Variablen vom Typ eines Aufzählungstyp deklariert werden

```
enum wochentage {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
Sonntag};

int main(void) {
   enum wochentage w1 = Mittwoch;    /* w1 hat den Wert 2 */
   enum wochentage w2 = 77           /* Funktioniert auch, w2 hat den Wert 77
}
```

- Kombination mit typedef, um enum bei der Variablendeklaration nicht schreiben zu müssen

```
typdef enum {Montag, Dienstag, …} Wochentage;

int main(void) {
   Wochentage w1 = Mittwoch;
}
```

3.2 Strukturen

- Erlaubt Zusammenfassung von Elementen verschiedener Typen in einen Datentyp
  - Entspricht ungefähr einer Java Klasse, bei welcher alle Variablen public sind und die keine Methoden enthält
- Strukturen werden auf dem Stack abgelegt
- Wenn ein int 4 Bytes beansprucht, so benötigt die Struktur im unteren Beispiel 3 * 4 = 12 Bytes
- Zuerst muss der neue Datentyp deklariert werden (meist am Anfang des Programms)

```
struct point3D {
   int x;
   int y;
   int z;
};
```

- Danach können Variablen dieses Typs deklariert / initialisiert werden

```
struct point3D pt = {2, 4, 6};
```

- Strukturen können auch geschachtelt werden

```
struct line3D {
   struct point3D start;
   struct point3D end;
}
```

- Zugriff auf einzelne Elemente (members) geschieht mit "."
- Wie alle Variablen können Variablen, die den Datentyp der gleichen Struktur aufweisen, einander zugewiesen werden

```
/* point3d.c */
#include <stdio.h>

/* Deklaration der Struktur */
struct point3D {
   int x, y, z;
};

int main(void) {
   struct point3D ptA = {2, 4, 6}, ptB;
   ptB = ptA;      /* Kopiert die komplette Struktur */
   ptA.x = 5;      /* Zugriff auf einzelne Elemente */
   ptA.y += ptA.z;

   /* Output: A = (5, 10, 6), B = (2, 4, 6) */
   printf("A = (%d, %d, %d)\n", ptA.x, ptA.y, ptA.z);
   printf("B = (%d, %d, %d)\n", ptB.x, ptB.y, ptB.z);
}
```

- Der Gebraucht mit struct point3D pt ist relativ mühsam, weil man das struct-Keyword immer schreiben muss
- In der Praxis wird man die Struktur deshalb oft gleich auch als neuen Datentyp definiert
  -> mit typedef möglich

```
typedef struct {
      int x, y, z;
} Point3D;
```

- Anschliessend kann man Point3D wie einen Datentypen behandeln; die Deklaration einer Variablen geschieht entsprechend wie folgt

```
Point3D pt = …    /* statt struct point3D pt = … */
```

4 Funktionen

- Erlauben die Strukturierung von Programmen
- Programmteile, die mehrmals in einem Programm vorkommen, müssen nur einmal implementiert werden
- Funktionen in C entsprechen in etwa den Methoden in Java
- Funktionen bestehen aus
  - Funktionsname (identifiziert Funktion eindeutig)
  - Parameterliste (mehrere Parameter mit Komma getrennt; void bzw. leer, wenn keine Parameter übergeben werden)
  - Datentyp des Rückgabewerts (void, wenn nicht zurückgegeben wird)

```
Datentyp_des_Rückgabewerts Funktionsname(Parameterliste)
```

- Parameter besteht aus Typ und Namen
  - Mit dem Namen kann der Parameter innerhalb der Funktion angesprochen werden

```
int max(int a, int b)
```

- Funktion mit dem Namen max
- Zwei Parameter vom Typ int, Namen: a und b
- Funktion gibt einen int-Wert zurück
- Bei der Funktionsdefinition wird die eigentliche Funktion genau einmal im Programm implementiert

```
int max(int a, int b) {
   if(a >= b) {
      return a;
   }
   return b;
}
```

- Zusätzlich wird die Funktionsdeklaration vor dem ersten Aufruf der Funktion im Programm benötigt

```
int max(int a, int b);
```

- Typischerweise: Funktionsdeklaration am Anfang eines Programms, gleich nach #include
- Ist Quellcode auf mehrere Dateien verteilt, dann muss die Funktionsdeklaration in jeder Datei, in welcher die Funktion aufgerufen wird, erfolgen (besser: in Header-Datei)

4.1 Aufruf einer Funktion

- Beim Funktionsaufruf müssen für sämtliche Parameter Werte übergeben werden
  - Meist mittels Variablen; Übergabe von Literalen ist auch möglich
  - Reihenfolge entspricht der Parameterliste in der Funktionsdefinition
- Hat Funktion einen Rückgabewert, so kann (muss aber nicht) dieser einer Variablen zugewiesen werden oder gleich in einem Audruck wiederverwendet werden

4.1.1 Beispiel

```
/* max.c */
#include <stdio.h>
int max(int a, int b);  /* Deklaration */

int main(void) {
   int x, y;
   printf(1. Zahl: ");
   scanf("%d", &x);
   printf("2. Zahl: ");
   scanf("%d", &y);
   printf("Maximum: %d\n", max(x, y);     /* Aufruf */

}

int max(int a, int b) { /* Definition */
   if(a >= b) {
      return a;
   }
   return b;
}
```

4.2 Parameter und Rückgabewerte

4.2.1 Als Parameter zugelassene Typen

- Grundlegende Datentypen (int, double, …)
- Strukturen
- Arrays
- Pointer

4.2.2 Als Rückgabewerte zugelassene Typen

- Grundlegende Datentypen (int, double, …)
- Strukturen
- Pointer

- Es können keine Arrays zurückgegeben werden, aber Pointer auf Datenbereiche, die einen Array enthalten

4.2.3 By value

- In C werden Parameter immer "by value" übergeben, d.h. die Werte der Variablen, die der Funktion übergeben werden, werden in die Funktion "hinein kopiert"
- Die Parameter können dann innerhalb der Funktion wie "normale" lokale Variablen verwendet werden
- Eine Veränderung dieses Wertes innerhalb der Funktion hat keinen Einfluss auf die Variable, die der Funktion übergeben wird

4.2.4 Return

- Mit return wird eine Funktion immer sofort verlassen
  - Hat die Funktion keinen Rückgabewert (void), so wird einfach return; ohne Rückgabewert geschrieben
- Hat die Funktion einen Rückgabewert, so wird return mit einem entsprechenden Wert geschrieben, z.B. return -1;

- Wird das Ende einer Funktion erreicht, so wird die Funktion auch ohne Angabe von return verlassen
  - hat die Funktion einen Rückgabewert, so wird der Default der Wert 0 zurückgegeben
- Die Rückgabe eines Wertes erfolgt "by value"

4.3 Funktionen ohne Parameter

- Funktionen können keine Parameter haben, man kann dies auf zwei Arten in der Deklaration / Definition angeben

```
int generateRandomInt(void);
int generateRandomInt();
```

- Die Variante mit void bezeichnet eine "wirklich leere" Parameterliste und der Compiler prüft dies beim Aufruf der Funktion
- Die Variante ohne void weist den Compiler an, Parameter-checking zu unterlassen
  - Ursprünglich gab es in C gar kein Parameter-checking

5 Lokale Variablen

- Neben den Parametern, welche einer Funktion übergeben werden, können lokale Variablen definiert werden

```
void printLine(int n) {
   int i;                        /* lokale Variable */
   for(i = 0; i < n; i++) {
      printf("_");
   }
   printf("\n");
}
```

- Nur innerhalb der Funktion sichtbar
- Entsteht kein Konflikt mit einer Variablen in einer anderen Funktion, die den gleichen Namen hat

```
int i = 15;       /* Nicht die gleiche Variable, wie die in der Funktion
                  printLine */
printLine(i);
```

- Die Variablen in der Funktion main sind auch lokale Variablen
  → alle innerhalb von Funktionen deklarierten Variablen sind lokale Variablen
- Generell sind lokale Variablen nur in dem Block sichtbar (verwendbar), in welchem sie deklariert wurden

```
int main(void) {
   int i;
   for(i = 0; i < 10; i++) {
      int j = 5;
      printf("%d\n", j);
   }
}
```

6 Globale Variablen

- Variablen, auf die alle Funktionen zugreifen können
- Werden ausserhalb von allen Funktionen definiert und müssen in jeder Funktion, die auf sie zugreift, mit **extern** deklariert werden

```
/* checkmax_glob.c */
#include <stdio.h>

int max = 0;            /* Definition glob. Variable */
void checkMax(int a);

int main(void) {
   int a;
   extern int max;      /* Deklaration glob. Variable */
   while(1) {
      printf("a = ");
      scanf("%d", &a);
      checkMax(a);
      printf("Maximum bis jetzt: %d\n", max);
   }
}

void checkMax(int a) {
   extern int max;      /* Deklaration glob. Variable */
   if(a > max) {
      max = a;
   }
}
```

7 Statische lokale Variablen

- Lokale Variablen, die mit static als statische Variablen deklariert werden, behalten ihren Wert auch nach dem Verlassen und Wiedereintritt in eine Funktion
- Statische Variablen werden nur einmal, gleich nach dem Programmstart initialisiert
- Statische Variablen erhalten per Default den Wert 0
- Oft lassen sich damit globale Variablen ersetzen

```
/* checkmax_stat.c */
#include <stdio.h>
int checkMax(int a);
int main(void) {
   int a;
   while(1) {
      printf("a = ");
      scanf("%d", &a);
      printf("Maximum bis jetzt: %d\n", checkMax(a));
   }
}
int checkMax(int a) {
   static int max = 0;  /* Deklaration statische lokale Variable*/
   if(a > max) {
      max = a;
   }
   return max;
}
```

8 Rekursive Funktionen

- Eine Funktion, die sich selbst wieder aufruft
- Bei jedem weiteren rekursiven Aufruf werden Parameter wieder "by value" übergeben, ebenfalls hat jede Rekursionsstufe eigene lokale Variablen

```
/* Rekursion */
int fakultaet(int n) {
  if(n < 2) {
    return 1;
  } else {
    return n * fakultaet(n-1);
  }
}
```

9 Arrays (und Strings)

- Enthält mehrere Elemente des gleichen Datentyps
  - Bei Deklaration werden Name, Datentyp und Anzahl Elemente angegeben
  - Index läuft von 0 … (Anzahl Elemente -1)
- Elemente eines Arrays der Länge n sind einfach direkt hintereinander im Speicher abgelegt
- Array mit 100 int-Elementen benötigt 400 Bytes Speicherplatz
- Speicherplatz wird statisch (zu Kompilierungszeit) auf dem Stack alloziert

```
int data[100]; /* Deklariert int Array mit 100 Elementen und Namen data */
```

- **Vorteil:** sehr effizient und flexibel im Gebrauch
- **Nachteil:** Array kennt seine Länge nicht (es gibt kein data.length), Zugriff über Arraygrenzen möglich
- Arrays können gleich bei der Deklaration Werte zugewiesen erhalten

```
int a[5] = {4, 7, 12, 77, 2}; /* Alle 5 Elemente werden initialisiert */
int b[] = [4, 7, 12, 77, 2};  /* Länge wird implizit auf 5 gesetzt */
int c[5] = {4, 3, 88, 5};          /* OK, letztes Element erhält Wert 0 */
int d[5] = {4, 3, 88, 5, 3, 6};    /* Kompilierfehler */
```

- Zugriffa uf Arrayelemente wie in Java

```
a[3] = 5;
b[2] = b[4] + a[0];
```

- Zur Laufzeit wird nicht geprüft, ob der Zugriff auf Arrayelement wirklich innerhalb des Arrays liegt

```
int a[5] = {4, 7, 12, 77, 2};
a[3] = 4;
a[8] = 222; /* Kein Fehler zur Laufzeit! */
a[-3] = 36; /* Kein Fehler zur Laufzeit! */
```

- Zuweisung mit {…} funktioniert nur bei Deklaration der Variablen.
  - Folgendes gibt einen Kompilierfehler

```
int a[5];
a = {1, 3, 66, 34, 7};  /* Fehler */
```

- Arrays können wie primitive Datentypen als konstant definiert werden
  - Werte können nach der Deklaration nicht mehr verändert werden
  - Initialisierung bei der Deklaration ist dabei natürlich sinnvoll, wird vom Compiler aber nicht erzwungen

```
int a[5] = {0, 1, 2, 3, 4};
const int b[5];   /* Funktioniert, macht aber kaum Sinn */
const int c[5] = {5, 6, 7, 8, 9};
const int d[] = {10, 11, 12, 13, 14};

a[0] = 4;         /* OK */
b[1] = 55;        /* Kompilierfehler */
c[2] = 666;       /* Kompilierfehler */
d[3] = 6666;      /* Kompilierfehler */
```

9.1 Mehrdimensionale Arrays

```
/* 2-dim Array mit 2 Zeilen und 3 Spalten */
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

a[0][0]	a[0][1]	a[0][2]
1	2	3
4	5	6
a[1][0]	a[1][1]	a[1][2]

- Zugriff auf Elemente durch spezifizieren beider Indizes

```
int b;
b = a[0][0];  /* b ist jetzt = 1 */
a[1][2] += 2 * b; /* a[1][2] ist jetzt = 8 */
a[3][3] = 7; /* Auch hier findet keine Überprüfung der Grenzen statt! */
```

9.2 Char-Arrays und Strings

- In C gibt es kein Typ string, ein String wird als char-Array dargestellt
  - nach dem letzten Zeichen im String folgt das Zeichen '\0' (NUL) (Ende des Strings)
- String Konstante (String literal) wird in "…" geschrieben
  - "Ich bin ein String" → 18 Zeichen
  - Interne Darstellung mit einem \0 am Ende → 19 Zeichen

I	c	h		b	i	n		e	i	n		S	t	r	i	n	g	\0

- Anwendung von String literals:
  - Direkt in einer Funktion: printf("hello, world");
  - Bei der Deklaration eines char-Arrays: char hello[] = "hello, world";
- Deklaration eines char-Arrays, Initialisierung mit String literal

```
char hello1[] = "hello, world";     /* Array mit 13 Zeichen */
```

h	e	l	l	o	,		w	o	r	l	d	\0

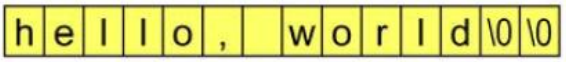
```
char hello2[13] = "hello, world";   /* OK, 13 Zeichen */
```

h	e	l	l	o	,		w	o	r	l	d	\0

```
char hello3[12] = "hello, world";   /* OK in C, aber \0 fehlt! */
```

h	e	l	l	o	,		w	o	r	l	d

```
char hello4[14] = "hello, world";   /* OK, mit \0, weitere Arrayelemente
werden per Default ebenfalls auf \0 initialisiert */
```



- Deklaration eines char-Arrays ohne Initialisierung

```
char ca1[20];           /* char-Array für 20 chars, hat noch nicht mit
                        einem String zu tun! */
char ca2[];             /* Kompilierfehler */
ca1 = "hello, world";   /* Kompilierfehler */
ca1[0] = 'h';
ca1[1] = 'e';
…
ca1[11] = 'd';
ca1[12] = '\0'; /* Jetzt kann man sagen, in ca1 ist ein String abgelegt! */
```

- Beispiel mit Strings und printf

```
/* string_printf.c */
#include <stdio.h>

int main(void) {
  char a[] = "Hans";
  char b[5];

  printf("%s\n", a);    /* String in a wird ausgegeben -> "Hans", printf
                        erkennt \0 am Ende */
  b[0] = 'H';
  b[1] = 'a';
  b[2] = 'n';
  b[3] = 's';
  printf("%s\n", b);    /* String in b wird ausgegeben -> "Hans", kein
                        \0 nach dem s -> printf gibt weiter Zeichen
                        aus bis irgendwann \0 im Speicher gefunden wird */
  b[4] = '\0';
  printf("%s\n", b);    /* \0 am Ende wird von printf erkannt */
```

9.3 Wichtige String Funktionen

Beschreibung	**Funktion**
Länge eines Strings (ohne abschliessendes '\0')	int strlen(const char s[]);
Vergleich zweier Strings (s1 > s2 → >0, s1 = s2 → 0, s1 < s2 → <0)	int strcmp(const char s1[], const char s2[]); Bsp: strcmp("Hans", "Haus") gibt einen Wert < 0 zurück
Kopieren eines Strings von source nach dest	char* strcp(char dest[], const char source[]);
Zusammenhängen zweier Strings (s2 wird an s1 angehängt)	char* strcat(char s1[, const char s2[]);

10 Pointer

- Bisher: Zugriff erfolgte immer direkt auf die Variablen; ohne Berücksichtigung wo (unter welcher Adresse) diese Variable im Speicher abgelegt ist
- Pointer ist eine Variable, welche eine Adresse enthält
- Bei Deklaration eines Pointers wird der Typ des Objekts, auf welches der Pointer zeigt, angegeben

```
int *p;     /* p ist ein Pointer auf ein Objekt vom Typ int */
int* p;     /* p ist ein Pointer auf ein Objekt vom Typ int */
double *d[20];    /* d ist ein Array von 20 Pointern auf Objekte vom Typ
                  double */
double (*d)[20];  /* d ist ein Pointer auf einen double Array mit 20
                  Elementen */
char **ppc; /* Pointer auf einen Pointer auf ein Objekt vom Typ char */
```

- Zwei Operatoren im Zusammenhang mit Pointern
  - Der Referenzoperator oder Adressoperator & liefert die Adresse eines Objekts
  - Der Dereferenzoperator oder Zugriffsoperator * kann auf Pointer angewendet werden, und liefert die Daten des Objekts, auf welches der Pointer zeigt

```
int i;
int *ip;    /* Pointer auf int */
ip = &i;    /* Zuweisung der Adresse von i; ip zeigt jetzt auf i */
*ip = 3;    /* ip wird dereferenziert und dem Objekt wird 3 zugewiesen;
            die Variable i ist jetzt also 3 */
```

10.1 void Pointer

- Pointer können vom Typ void sein; ein solcher Pointer kann auf irgendetwas zeigen

```
double d;
double *dp1 = &d;
void *vp = dp1;
```

- Um einen void Pointer in einen expliziten Typen zu konvertieren, wird in C++ ein cast benötigt (optional in C)

```
double *dp2;
dp2 = (double *) vp;
```

- void Pointer sind nützlich, um mit einem Pointer einen Block von Daten zu referenzieren, ohne zu wissen, was in den Daten drinsteht oder wozu der Datenblock gebraucht wird
- **malloc** (um dynamischen Speicher zu allozieren) liefert z.B. einen void Pointer auf einen Datenbereich, denn malloc weiss nicht, wofür dieser Datenbereich gebraucht wird

10.2 Pointer und const

```
double *cdp;      /* Pointer auf double, *cdp = 5.0 und cdp++ möglich */
double *const cdp;      /* Konstanter Pointer, *cdp = 5.0 möglich, nicht
                        aber cdp++ */
const double *dp;       /* Pointer auf Konstante, cdp++ möglich,
                        aber nicht *cdp = 5.0 */
const double *const dp; /* Konstanter Pointer auf Konstante, cdp++ und
                        *cdp = 5.0 nicht möglich */
```

10.2.1 Beispiele

```
const double pi = 3.1415    /* Konstante pi, Wert kann nicht geändert werden
double *dp = &pi;           /* Fehler, vermeiden von *dp = 5.0 */
double *const dp = &pi;     /* Fehler, vermeiden von *dp = 5.0 */
const double *dp = &pi;     /* OK, Pointer auf Konstante, *dp = 5.0 nicht
                               mehr möglich */
const double *const dp = &pi; /* OK, Konstanter Pointer auf Konstante */
```

- Steht ein const "rechts vom *", so ist der Pointer konstant
- Steht ein const "links vom *", so ist das Objekt, auf welches der Pointer zeigt, konstant

10.3 Pointer und NULL

- Die Adresse 0 ist niemals eine gültige Speicheradresse
  - Wird deshalb oft für einen Pointer verwendet, der explizit "auf nichts" zeigt
  - stdio.h definiert dafür symbolische Konstante NULL
  - Gibt eine Funktion einen Pointer zurück, wird der Rückgabewert NULL zudem oft verwendet, um abnormales Verhalten zu signalisieren

```
/* null_pointer.c, gibt drei Zeilen aus */
#include <stdio.h>

int main(void) {
   int *p1 = 0;    /* das funktioniert */
   int *p2 = NULL;       /* das ist aber sauberer */

   if(p1 == NULL) printf("p1 ist NULL\n");
   if(p2 == 0) printf("p2 ist 0\n");
   if(p1 == p2) printf("p1 ist gleich p2\n");
}
```

10.4 Arrays und Pointer

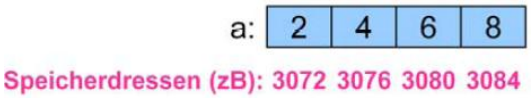
- Klarer Zusammenhang zwischen Arrays und Pointer
  - Namen des Arrays stellt fixe Startadresse des Arrays dar, diese Adresse kann nicht verändert werden
  - Wird ein Array in einem Ausdruck verwendet, so wird er vom Compiler immer implizit in den entsprechenden Pointer konvertiert

10.4.1 Beispiel

```
int a[3] = {2, 4, 6};
int b[3] = {2, 4 ,6};
int *pa;
a == b;     /* false, weil Startadressen von a und b unterschiedlich */
a = b       /* Kompilierfehler, weil Startadresse nicht veränderbar */
pa = a;     /* OK, a wird in int* const konvertiert -> pa zeigt jetzt auf
               den Array a; genauer: auf das erste Element a[0] */
```

- Zugriff auf Elemente eines Arrays bisher: Verwendung der Indizes

```
int a[4] = {2, 4, 6, 8};
int b = 3;
a[0] = a[1] + a[2];
a[3] = b;
b = a[2];
```



- In C wird ein Array im Speicher immer linear abgespeichert (Elemente folgen nacheinander)
  → auch mit Pointern kann auf die einzelnen Elemente zugegriffen werden
- Einen Fall kennen wir bereits, den Zugriff auf das erste Element

```
int a[3] = {2, 4, 6};
int *pa = a;      /* pa zeigt jetzt auf a und damit auf a[0] */
*pa = 7;          /* entspricht a[0] = 7; a[0] ist jetzt 7 */
```

10.5 Pointerarithmetik

- Ausser den bereits eingeführten speziellen Pointeroperatoren (*und &) sind diverse weitere Operatoren auf Pointer anwendbar
  - Vergleichsoperatoren: ==, !=, <, >, <=, >=
  - Addition + und Subtraktion -
  - Inkrement ++, Dekrement -- und zusammengesetzte Operationen +=, -=
- Regel: ist pa ein Pointer auf das erste Element eines Arrays, so zeigt (pa + i) auf das i-te Element dieses Arrays
- Beispiel

```
int a[5] = {2, 4, 6, 8, 10};
int *pa = a;
```

- Diese Ausdrücke sind äquivalent und bezeichnen das Arrayelement mit Index 2
  - a[2];
  - *(pa + 2);
  - *(a + 2);
  - pa[2];
- Damit Pointerarithmetik richtig funktioniert, wird der Datentyp des Pointers berücksichtigt

```
int a[5] = {2, 4, 6, 8, 10};
int *pa = a;      /* pa zeigt auf das erste Element im Array */
int *pb = pa;     /* pb zeigt auch auf das erste Element im Array */
pa++;             /* pa zeigt auf das nächste Element im Array, dabei wurde
                  der Wert von pa typischerweise um 4 inkrementiert
                  (Speicherplatz eines int) und nicht einfach nur um 1 */
pb += 3;          /* pb zeigt auf das Element mit Index 3 im Array wodurch
                  der Wert von pb um 3 * 4 = 12 inkrementiert wurde */
```

- Die Adressen einzelner Elemente im Array können mit dem Referenzoperator auch direkt einem Pointer zugewiesen werden

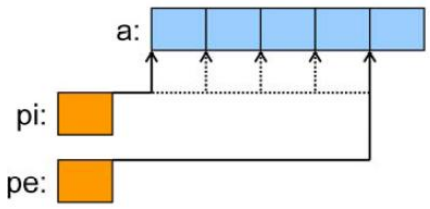
```
int a[5] = {2, 4, 6, 8, 10};
int *pa = &a[2];
*(pa + 2) = 13;   /* a[4] ist jetzt = 13 */
*(pa - 2) = 20;   /* a[0] ist jetzt = 20 */
```

- Diese Programme sind alle äquivalent

```
int main(void) {
   int a[5];
   int i;

   for(i = 0; i < 5; i++) {
       a[i] = 0;   /* oder *(a + i) = 0; */
   }
}
```

```
int main(void) {
   int a[5];
   int *pi = a;
   int i;

   for(i = 0; i < 5; i++) {
       *(pi + i) = 0; /* oder pi[i] = 0; */
   }
}
```

```
int main(void) {
   int a[5];
   int *pi = a;
   int *pe = &a[4];

   for(; pi <= pe; pi++) {
       *pi = 0;
   }
}
```



10.6 String Literals und Pointer

- Man kann Arrays und Pointer mit einem String Literal initialisieren

```
char a[] = "hello, world";
char *pa = "hello, world";
```

- Fast identisch, aber wichtiger Unterschied:
  - a ist ein char-Array und stellt fixe Startadresse dar → a kann nie auf einen anderen Speicherbereich zeigen
  - pa ist eine Pointervariable, die die Startadresse des char-Arrays enthält → kann später auch einen anderen Wert erhalten

10.6.1 Beispiel

```
char a[] = "hello, winterthur";
char *pa = "hello, switzerland";
a = pa;     /* Kompilierfehler */
pa = a;     /* OK, zeigt auf "hello, winterthur" */
```

10.7 Zwei-Dimensionale Arrays und Pointer

- Regeln für Beziehung von 1-Dimensionalen Arrays und Pointern
  - Name des Arrays stellt fixe Startadresse des Arrays dar, diese Adresse kann nicht verändert werden
  - Wird Array in Ausdruck verwendet, so wird er implizit in den entsprechenen Pointer konvertiert
- Basierent auf Regeln der Pointerarithmetik gilt

```
int a[5] = {1, 2, 3, 4, 5};
int *pa = a;

/* Alle folgenden Ausdrücke sind äquivalent */
a[3];
pa[3];
*(a + 3);
*(pa + 3);
```

- Diese Array-Pointer Beziehung gilt auch bei 2-Dimensionalen Arrays
- Repetition: Datentypen im Zusammenhang mit 1-Dimensionalem Array

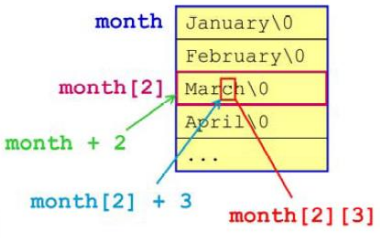
```
char module[4] = "CPP";
```

- Betrachtung eines 2-Dimensionalen Array, das die Monatsnamen speichert

```
char month[12][10] = {"January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};
```

- Daraus resultierendes Array braucht Speicherplatz von 12 * 10 = 120 Bytes
- Die 12 einzelnen Arrays mit den Monatsnamen sind nacheinander im Speicher angeordnet
  - Zuerst 10 Zeichen für January\0, wovon 2 Zeichen nicht gebraucht werden
  - dann 10 Zeichen für February\0 etc.
- Adresse von 'F' in February ist 10 Bytes grösser als Adresse von 'J' in January
- month hat den Datentyp char[12][10]
- month[2] ist vom Typ char[10]
- month[2][3] hat Datentyp char
- month[2] + 3 vom Typ char*
- month + 2 ist vom Typ char(*)[10]

- Die nebenstehende Abbildung zeigt, welche Bereiche der Tabelle durch die Ausdrücke identifiziert werden
- Daraus sieht man, dass folgende Zusammenhänge gelten:
  - `*(month + 2) = month[2]`
  - `*(month[2] + 3) = month[2][3]`



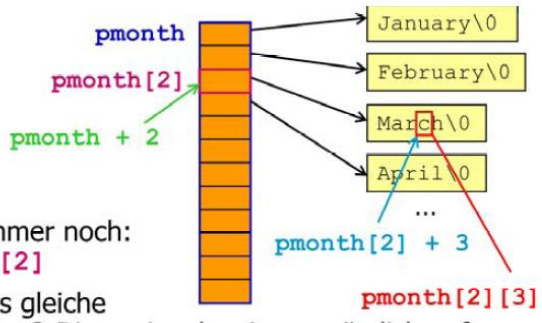
- Dies „geht auch auf", wenn die Datentypen betrachtet werden:
  - `month + 2`: `char(*)[10]`, `month[2]`: `char[10]`
  - `month[2] + 3`: `char*`, `month[2][3]`: `char`
- Daraus folgt, dass bei 2-Dimensionalen Arrays die Array-Pointer-Beziehung sowohl in der ersten als auch in der zweiten Dimension gilt
- Treibt man dies weiter: generell gilt die Beziehung für alle Dimensionen eines n-Dimensionalen Arrays

- Das Speichern von mehreren Strings (oder Arrays verschiedener Längen im Allgemeinen) als 2-Dimensionaler Array ist nicht effizient, weil jeder einzelne String soviel Speicher wie der längste dieser Strings braucht
- Besser: Speichern der einzelnen Strings als 1-Dimensionale Arrays "irgendwo" und merken uns deren Speicherstellen in einem Array von Pointern

```
char* pmonth[12] = {"January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"};
```

- Die 12 Elemente in pmonth (Startadressen der Strings) sind immer noch linear nacheinander im Speicher angeordnet
- Es gibt jedoch keine Garantie, dass auch die einzelnen Strings selbst nacheinander im Speicher liegen

- Wiederum bestimmen wir die Datentypen:
  - `pmonth` hat den Datentyp `char*[12]` (gemäss Deklaration) und ist ein 1-Dimensionaler Array mit 12 Elementen vom Typ **`char*`**
  - `pmonth[2]` greift auf eines der 12 Arrayelemente in **`pmonth`** zu (auf das mit Index 2) und ist deshalb vom Typ `char*`
  - `pmonth[2] + 3`: **`pmonth[2]`** ist vom Typ **`char*`**, also ist **`pmonth[2] + 3`** ebenfalls vom Typ `char*`
  - `pmonth + 2`: **`pmonth`** ist vom Typ **`char*[12]`** (1-D Array mit **`char*`** Elementen) und wird in einem Ausdruck verwendet, also ist **`pmonth + 2`** vom Typ `char**` (Pointer auf einen Pointer auf einen **`char`**)



- Wiederum zeigt die Abbildung, welche Bereiche der Tabelle durch die Ausdrücke identifiziert werden
- Die Array-Pointer-Beziehung in der ersten Dimension gilt immer noch: **`*(pmonth + 2) = pmonth[2]`**
- `pmonth[2] + 3` zeigt auf das gleiche Zeichen wie `month[2] + 3` im 2-Dimensionalen Array, nämlich auf das Zeichen c in March und hat auch den gleichen Datentyp (**`char*`**)
  - Wir wissen: beim 2-Dimensionalen Array ist wegen der Array-Pointer-Beziehung **`*(month[2] + 3)`** dasselbe wie **`month[2][3]`**
  - Ein C-Compiler wandelt Ausdrücke wie **`month[2][3]`** konsequent in **`*(month[2] + 3)`** um, deshalb kann auch `pmonth[2][3]` geschrieben werden, um das Zeichen c in March zu bezeichnen
- Fazit: man kann auch mit einem Array von Pointern auf Arrays so tun, als ob es ein 2-Dimensionaler Array wäre und mit der „gewohnten Schreibweise" auf Elemente zugreifen: **`pmonth[2][3]`**

10.8 Strukturen und Pointer

- Oft verwendet man Pointer auf Strukturen; dabei gibt es eine abgekürzte Schreibweise (->), um auf Elemente in der Struktur zuzugreifen

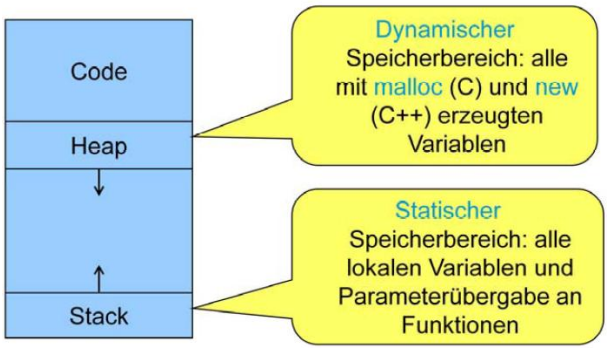
```
struct student {
   int studNr;
   char name[30];
   char vorname[30];
};
struct student *sp, s;  /* sp ist Pointer auf Struktur student */
s.studNr = 999;         /* Initialisierung von s */
strcpy(s.name, "Mueller");
strcpy("s.vorname, "Max");
sp = &s;                /* sp zeigt auf die Struktur s */
printf("Student: %s %s, Nr. %d\n", (*sp).name, (*sp).vorname,
            (*sp).studNr);
/* oder besser (wegen Lesbarkeit) */
printf("Student: %s %s, Nr. %d\n", sp->name, sp->vorname, sp->studNr);
```

- Eine Struktur kann mit malloc auch dynamisch auf dem Heap alloziert werden

```
struct *sp;
sp = (struct student*) malloc(sizeof(struct student));
```

11 Statisch und dynamisch allozierter Speicher

- Default: Speicherplatz wird statisch alloziert (auf dem Stack)
- Speicherplatz kann aber auch dynamisch (auf dem Heap) alloziert werden
- Speicherorganisation (aus Sicht eines ausführenden Programms)



- Dynamisch allozierter Speicher wird zur Laufzeit auf dem Heap mit der Standard Library Funktion malloc erzeugt

```
void *malloc(int size)
```

- Wird prinzipiell dann gebraucht, wenn zur Kompilierungszeit nicht bekannt ist, wieviel Speicher zur Laufzeit benötigt wird
- malloc macht zwei Dinge
  - alloziert Speicher der Grösse size Bytes auf dem Heap
  - gibt die Adresse (also einen Pointer) dieses Speicherplatz zurück (void *)
    → Pointer muss noch in den gewünschten Datentyp konvertiert werden

- Beispiel

```
int *ip;
ip = (int *) malloc(sizeof(int));
*ip = 5;
```

- Der sizeof Operator liefert die Grösse des entsprechenden Datentyps
- Dynamisch allozierter Speicher wird **nicht automatisch** freigegeben (statischer schon), wenn eine Variable nicht mehr benötigt wird (z.B. Verlassen einer Funktion)
- Dynamisch allozierter Speicherplatz wird mit free freigegeben

```
void free(void *vp)
```

- Beispiel

```
int *ip;
ip = (int *) malloc(sizeof(int));
*ip = 5;
...
free(ip);   /* Speicher wird freigegeben */
```

- Ein Unterlassen der Speicherfreigabe kann langfristig das Programm zum Absturz bringen (Memory Leaks!)
- Man kann auch dynamisch Speicher für einen Array allozieren
- Beispiel: Array für 250 int-Werte

```
int *ap;
int n = 250;
ap = (int *) maloc(n * sizeof(int));
```

- Alloziert 1000 Bytes
- Diese 1000 Bytes liegen hintereinander auf dem Heap, Speicherstellen x … (x + 999)
- ap zeigt auf den Anfang dieses Bereichs
  → entspricht im Wesentlichen der internen Darstellung eines Arrays

```
/* zahlen1.c */
#include <stdio.h>


int main(void) {
  int n, i, numbers[100];

  printf("Anzahl Zahlen: ");
  scanf("%d", &n);


  for (i = 0; i < n; i++) {
    printf("%d. Zahl: ", (i + 1));
    scanf("%d", &numbers[i]);
  }
  printf("Die Zahlen sind:");
  for (i = 0; i < n; i++) {
    printf(" %d", numbers[i]);
  }
  printf("\n");

}
```

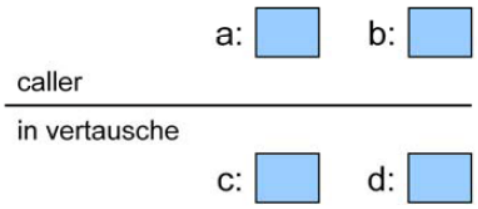
```
/* zahlen2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int n, i, *numbers;

  printf("Anzahl Zahlen: ");
  scanf("%d", &n);
  numbers = (int *)
            malloc(n*sizeof(int));
  for (i = 0; i < n; i++) {
    printf("%d. Zahl: ", (i + 1));
    scanf("%d", &numbers[i]);
  }
  printf("Die Zahlen sind:");
  for (i = 0; i < n; i++) {
    printf(" %d", numbers[i]);
  }
  printf("\n");
  free(numbers);
}
```

12 Funktionen 2

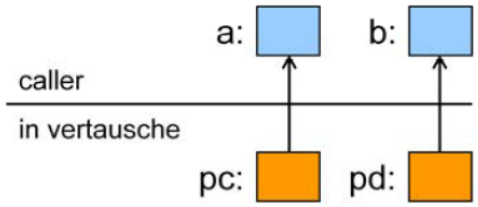
12.1 Call by address

- Beispiel: Funktion vertausche, die die Werte zweier Variablen vertauschen soll

```
/* call by value (funktioniert NICHT) */
int a = 3; b = 5;
vertausche(a, b);

void vertausche(int c, int d) {
   int temp = c;
   c = d;
   d = temp;
}
```




```
/* call by address (funktioniert) */
int a = 3; b = 5;
vertausche(&a, &b);

void vertausche(int *pc, int *pd) {
   int temp = *pc;
   *pc = *pd;
   *pd = temp;
}
```



12.2 Arrays als Parameter

- Wird ein Array in einem Ausdruck verwendet, so wird er implizit in den entsprechenden Pointer konvertiert
- Wird ein Array einer Funktion übergeben, wird implizit ein Pointer auf diesen Array übergeben
  →Arrays werden immer "by address" übergeben
- Weiter spielt es keine Rolle, ob einer Funktion vom Programmierer explizit ein Array oder ein Pointer auf einen Array übergeben wird; für die Funktion ist es immer ein Pointer
- Ebenso kann in der Parameterliste ein Parameter entweder als Array oder als Pointer spezifiziert werden

```
rückgabewert funcName(int a[]);
rückgabewert funcName(int *a);
```

- Beispiel: beide Funktionen sind äquivalent (Berechnen des Skalarprodukts zweier Arrays)

```
double sprod(double first[], double second[], int len) {
   double sum = 0.0;
   int i;

   for(i = 0; i < len; i++) {
       sum += first[i] * second[i];
   }
   return sum;
}
```

```
double sprod(double *first, double *second, int len) {
   double sum = 0.0;
   int i;

   for(i = 0; i < len; i++) {
       sum += *first++ * *second++;
   }
   return sum;
}
```

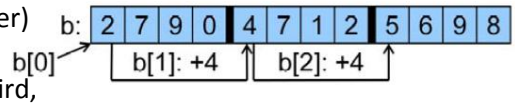
- Die Parameterübergabe beim Funktionsaufruf kann ebenfalls mit Arrays oder mit Pointern erfolgen, unabhängig davon, welche der beiden Funktionen oben implementiert wurde

12.3 2-Dimensionale Arrays als Parameter

- Wird ein 2-Dimensionales Array a[3][4] einer Funktion übergeben, so muss in der Parameterliste die zweite Dimension spezifiziert werden

```
rückgabewert funcName(int b[][4]);
rückgabewert funcName(int b[][]);   /* FALSCH, Kompilierfehler */
```

- Die zweite Dimension ist notwendig, damit in der Funktion Anweisungen wie b[2] korrekt ausgeführt werden können
  - Die Angabe der zweiten Dimension ermöglicht es herauszufinden, wo die einzelnen Arrays im Array beginnen (wegen der linearen Anordnung im Speicher)
  - Ohne die Angabe dieser zweiten Dimension wüsste der Compiler nicht, wie auf die einzelnen Arrays zugegriffen wird, denn bei der Parameterübergabe wird nur ein Pointer auf b übergeben



- Bei der Konvertierung des 2-Dimensionalen Arrays a[3][4] in einen Pointer wird der Datentyp zu int (*)[4] (Pointer auf einen int Array der Länge 4)
- Deshalb sind folgende Funktionsdeklarationen äquivalent

```
rückgabewert funcName(int b[][4]);
rückgabewert funcName(int (*b)[4]);
```

- Auch hier können bei beiden Varianten Variablen vom Typ int(*)[4] und int[][4] übergeben werden
- Vergleich mit einem 1-Dimensionalen Array a[4]

```
rückgabewert funcName(int b[]);
rückgabewert funcName(int *b);
```

- Generell: Bei der Übergabe von Arrays an Funktionen müssen in der Parameterliste alle Dimensionen ausser der ersten angegeben werden

12.4 Array of Pointers als Parameter

- Wird ein Array of Pointer, z.B. int *a[10] einer Funktion übergeben, ist dies nicht anderes als die Übergabe eines 1-Dimensionalen Arrays

```
rückgabewert funcName(int *b[]);
```

- Bei der Konvertierung in einen Pointer wird der Datentyp zu int **b (Pointer auf einen Pointer auf einen int)

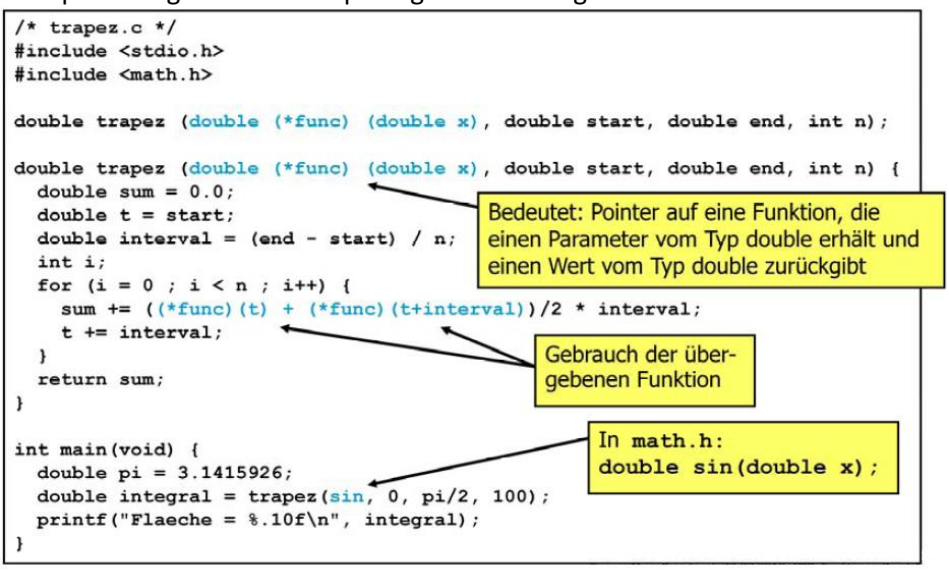
- Deshalb ist diese Funktionsdeklaration äquivalent zu der oben stehenden

```
rückgabewert funcName(int **b);
```

- Wiederum können unabhängig davon, welche Funktionsdeklaration verwendet wird, Variablen der Typen int*[10] und int** übergeben werden

12.5 Funktionen als Parameter

- Man kann einer Funktion auch einen Pointer auf eine Funktion übergeben
- Beispiel: Integration mit Trapezregel für beliebige Funktionen



12.6 Array als Rückgabewert

- Der Rückgabewert einer Funktion **kann kein Array** sein, sehr wohl aber ein Pointer auf ein Array
- Beispiel: Funktion, die drei int-Parameter erhält, daraus einen int-Array der Länge 3 bildet und einen Pointer auf diesen Array zurückgibt

```
/* FUNKTIONIERT NICHT! getarray1.c */
int *getArray(int a, int b, int c) {

   int ret[3];

   ret[0] = a;
   ret[1] = b;
   ret[2] = c;
   return ret;
}
```

```
/* getarray2.c */
int *getArray(int a, int b, int c) {

   int *ret = (int *) malloc(3 * sizeof(int));
   *ret = a;
   *(ret + 1) = b;
   *(ret + 2) = c;
   return ret;
}
```

13 Konstante Parameter

- Ein Array wird immer by address einer Funktion übergeben; der Programmierer kann nicht wählen, den Array by value zu übergeben
  - es gibt keine lokale Kopie des Arrays innerhalb der Funktion
  - innerhalb der Funktion wird mit dem gleichen Array gearbeitet, wie ausserhalb (Zugriff auf die gleichen Speicherstellen)
- Dadurch hat jede Modifikation der Arrayelemente in der Funktion direkten Einfluss auf den Array, welcher der Funktion "übergeben" wurde
  - Potentielles Problem: wie kann der Programmierer z.B. sicher sein, dass eine Funktion einer Library den Array nicht modifiziert?
- Mittels Spezifizieren eines Parameters mit const stellt der Compiler sicher, dass der entsprechende Parameter innerhalb der Funktion nicht modifiziert wird
- Beispiel: Funktion, die eine Zahl, die als String dargestellt ist, in einen int konvertiert; der String darf in der Funktion nicht verändert werden

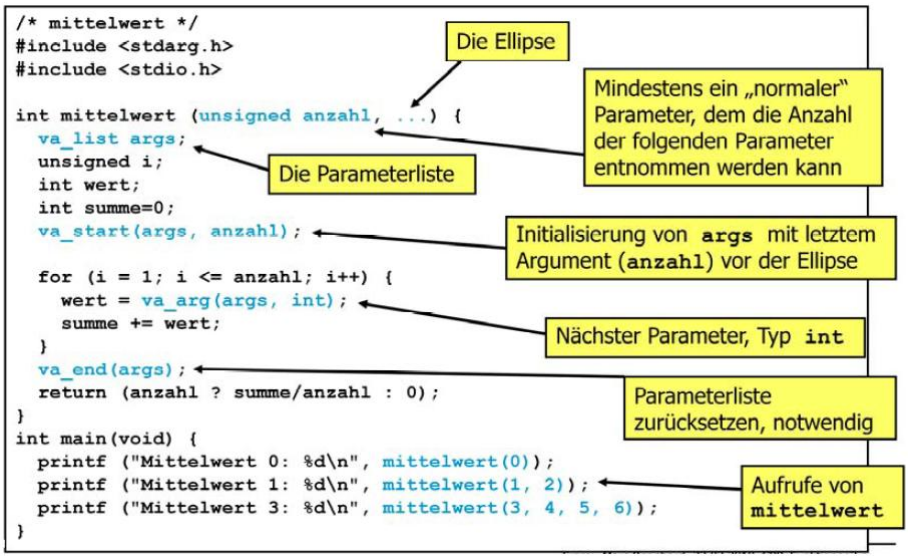
```
/* Parameter als konstanter char-Array */
int stringToInt(const char s[]) {
   int i, res = 0;

   for(i = 0; s[i] != '\0'; i++) {
       res = res * 10 + s[i] - '0';
   }
   return res;
}
```

```
/* Parameter als Pointer auf einen konstanten char */
int stringToInt(const char* s) {
   int res = 0;

   for(; *s != '\0'; s++) {
       res = res * 10 + *s - '0';
   }
   return res;
}
```

- Der Gebrauch von const in der Parameterliste (wenn sinnvoll) ist guter Programmierstil

14 Funktionen mit variabler Anzahl Parameter

- C erlaubt die Definition von Funktionen mit einer variablen Anzahl Parameter (kommt z.B. in printf und scanf vor)
- Die Implementierung dieser Funktionen beruht auf zwei Elementen
  - Der Ellipse in der Parameterliste
    - Die Ellipse … zeigt an, dass noch eine beliebige Anzahl Parameter kommen können
    - Die Ellipse muss am Ende der Parameterliste stehen
    - Ihr muss ein normaler Parameter vorangehen, an dem man innerhalb der Funktion erkennen kann, wie viele Parameter folgen
  - Den va_ Makros aus der Standard Library (stdarg.h); sie werden benötigt, um die Parameter zu übernehmen und auszuwerten
- Beispiel: Die Funktion mittelwert berechnet den Mittelwert einer variablen Anzahl von int-Parametern



15 Kommandozeilen Parameter

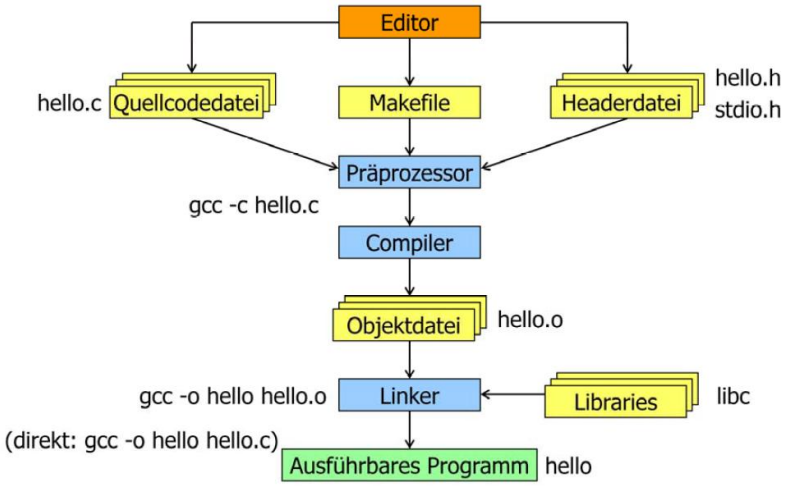
- Bisher haben wir immer int main(void) als Einstiegspunkt in eine Funktion verwendet
- Mit int main(int argc, char* argv[]) können Kommandozeilen Parameter an das Programm übergeben werden:
  - argc beschreibt die Anzahl Parameter
  - argv ist ein Array, das die Pointer auf die Kommandozeilen Parameter (Strings) enthält;
    - argv[0] ist dabei immer der Programmname
- Beispiel: Ausgabe des Namen des Programms und der Kommandozeilenparameter

```
/* commandline.c */
#include <stdio.h>
int main(int argc, char* argv[]) {
   int i;
   for(i = 0; i < argc; i++) {
       printf("%s\n", argv[i]);
   }
}
```

16 Modulare Programmierung

16.1 Präprozessor, Compiler und Linker

- Um aus Quellcode ein lauffähiges Programm zu generieren, sind drei Komponenten involviert
  - Der Präprozessor sucht im Quellcode nach speziellen Befehlen (z.B. #include, #define, #ifdef, …), um dann im Wesentlichen textuelle Ergänzungen durchzuführen
- Der Compiler wandelt dann den Quellcode in Objektcode um
  - Aus einer Quellcodedatei wird eine Objektcodedatei generiert
  - Die einzelnen Objektcodedateien können noch offene Aufrufe enthalten
    - z.B. auf Funktionen, die nicht innerhalb der eigenen Quellcodedatei definiert sind
    - oder auf globale Variablen in einer anderen Quellcodedatei
- Der Linker verbindet die noch offenen Aufrufe und generiert ein ausführbares Programm



- Mit gcc -S hello.c kann Assembler-Code generiert werden; Assembler ist im Wesentlichen eine "für den Menschen" lesbare Version von Objektcode

16.2 Präprozessor

16.2.1 Aufgaben des Präprozessors

- Daten von anderen Dateien in den Quelltext einbinden (#include)
- Mit #define definierte Konstanten durch die entsprechenden Werte zu ersetzen
- Selektiv Teile des Quellcodes ein- oder auszuschliessen (mit den Befehlen #if, #elif, #else, #endif, #ifdef, #ifndef)
- Präprozessorbefehle beginnen immer mit #
- Mit gcc -E kann der Output nach der Präprozessorstufe betrachtet werden
- Es gibt zwei Arten um Daten von anderen Dateien einzubinden
  #include <Dateiname> um Headerdateien der Standard Library einzubinden, z.B. #include <stdio.h>
  #include "Dateiname" um projektspezifische (d.h. selbst generierte) Headerdateien einzubinden, z.B. #include "hello.h"
- Konstanten werden mit #define definiert
  - #define MAXLOOP 1000 → jedes Auftreten von MAXLOOP im Quellcode wird durch 1000 ersetzt
  - int counter = MAXLOOP; → int counter = 1000;
- Achtung: man kann mit #define auch Fehler einbauen
  #define K 10
  int K = 20 → daraus wird int 10 = 20; dies wird vom Compiler entdeckt weil 10 kein gültiger Variablenname ist */

- Konstanten können auch keinen Wert haben und können "un"-definiert werden; dies macht aber im Zusammenhang mit selektiver Einbindung von Code nur Sinn:
  #define DEBUG /* Definieren einer Konstante DEBUG */
  #undef DEBUG /* DEBUG existiert nicht mehr */
- Selektiv Code ein- oder ausschliessen; z.B. um ein Programm für Debugging oder Production Mode zu kompilieren

```
#define DEBUG /* Konstante; ohne zugehörigen Wert */
...
#if defined DEBUG /* ist eine Konstante DEBUG definiert? */
  printf("Programm Version 1 (Debuggin)\n");   /* ja */
#else
  printf("Programm Version 1 (Production)\n"); /* nein */
#endif
```

- Kurzform für #if defined: #ifdef
- Alle Präprozessor-Befehle
  - #if, #elif, #else, #endif, #ifdef, #ifndef

16.3 Compiler

- Der eigentliche Compiler erhält vom Präprozessor den überarbeiteten Quellcode und generiert Objektcode
- Der Objektcode ist bereits maschinennaher Code, der aber noch nicht ausführbar ist, da er noch offene Aufrufe enthalten kann
- Aufgaben des Compilers
  - Prüfung des Quellcodes auf syntaktische Korrektheit
  - Statische Typprüfung (überprüft die Datentypen bei der Zuweisung von Variablen)
  - Gibt Errors und Warnungen aus
  - Treten keine Errors auf (Warnungen sind erlaubt, sollten aber vom Programmierer genau untersucht werden), wird der Objektcode erzeugt

16.4 Linker

- Generiert ein vollständiges, ausführbares Programm aus:
  - dem vom Compiler generierten Objektcode
  - Code aus der Standard Library oder aus anderen Libraries (welche auch nichts anderes als Objektcode enthalten)
- Die wichtigste Aufgabe des Linkers ist die Auflösung der noch offenen Aufrufe
  - Es wird geprüft, ob die aus anderen Objektdateien oder aus libraries verwendeten Funktionen dort auch wirklich vorhanden sind
  - Es wird geprüft, ob mit extern deklarierte Variablen auch wirklich irgendwo definiert werden
  - Der verwendete Objektcode wird in einer ausführbaren Datei zusammengesetzt und die Funktionsaufrufe und die zugehörigen Funktionen werden zusammengehängt (die Adressen der Funktion werden bei den Aufrufen richtig gesetzt)

16.5 Modulare Programmierung

- Aufteilen des Quellcodes auf mehrere Module
  - Verschiedene Module werden in verschiedenen Dateien abgelegt
- Weiter werden ein oder mehrere Headerdateien verwendet
  - Enthalten z.B. Konstanten (#define), Funktionsdeklarationen und Strukturen
  →dadurch kommen die entsprechenden Daten nur einmal im Quellcode vor

- Beispiel: Einfaches Programm mit einem #define und einer Funktion doit

```
/* main.c */
#define MAX 1000
int doit(int b); /* Funktionsdeklaration */

int main(void) {
   return MAX + doit(10);
}

int doit(int b) { /* Funktionsdefinition */
   return MAX + b;
}
```

16.5.1 1. Schritt: Aufteilung in zwei Dateien (Module)

```
/* main.c */
#define MAX 1000
int doit(int b); /* Funktionsdeklaration */

int main(void) {
   return MAX + doit(10);
}
```

```
/* doit.c */
#define MAX 1000
int doit(int b); /* Funktionsdeklaration; optional, weil doit(…) in diesem
                    Modul nicht aufgerufen wird */
int doit(int b) { /* Funktionsdefinition */
   return MAX + b;
}
```

16.5.2 2. Schritt: Verwendung einer Headerdatei

```
/* header.h */
#define MAX 1000
int doit(int b); /* Funktionsdeklaration */
```

```
/* main.c */
#include "header.h" /* alles, was in header.h steht, wird hier eingefügt */

int main(void) {
   return MAX + doit(10);
}
```

```
/* doit.c */
#include "header." /* alles, was in header.h steht, wird hier eingefügt */

int doit(int b) { /* Funktionsdefinition */
   return MAX + b;
}
```

Vorteil: Konstanten und Funktionsdeklarationen nur einmal vorhanden

- Generieren des ausführb0aren Programmes:

```
gcc -o prog main.c doit.c
```

- Zuerst generiert der Präprozessor folgendes

Aus doit.c

```
int doit(int b); /* Funktionsdeklaration, aus header.h */

int doit(int b) { /* Funktionsdefinition */
   return 1000 + b;
}
```

Aus main.c

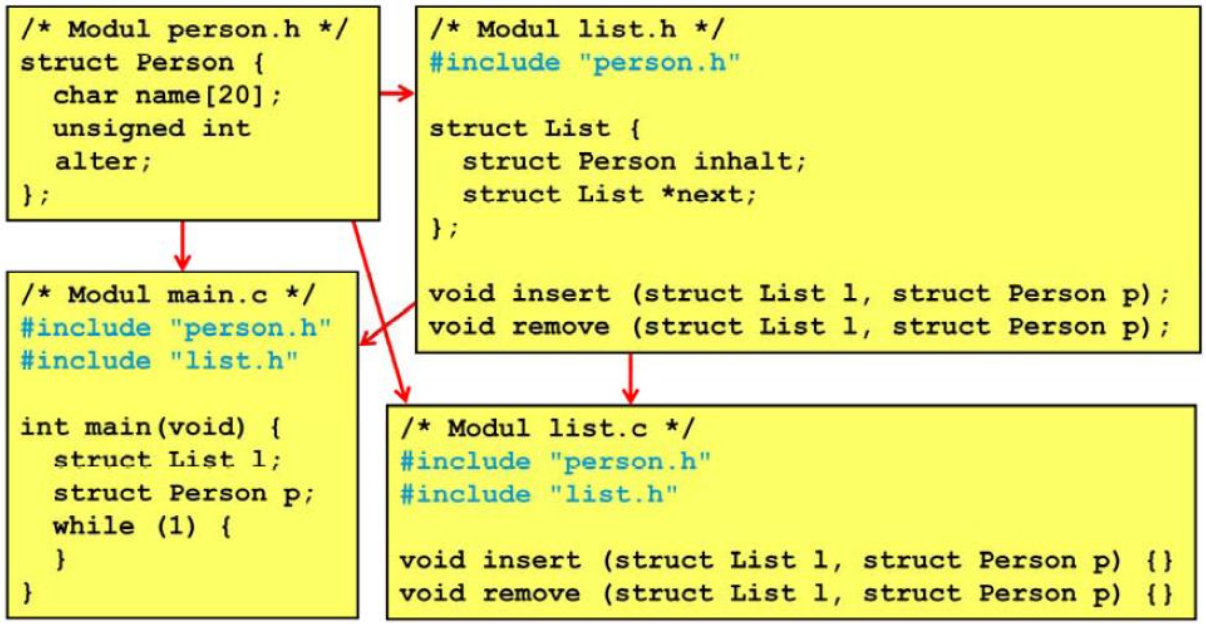
```
int doit(int b); /* Funktionsdeklaration, aus header.h */

int main(void) {
   return 1000 + doit(b);
}
```

- Anschliessend generieren Compiler und Linker das ausführbare Programm
- Einbinden von Headerdateien die nich tim aktuellen Verzeichnis liegen mit der Option -I

```
gcc -Idir -o prog main.c doit.c
```

16.5.3 Übung zu #include

- Wie sieht main.c nach dem Abarbeiten durch den Präprozessor aus?



- Problem: person.h wird einmal direkt (von main.c) und einmal indirekt (via list.h) included
- Die Struktur Person ist damit in main.c zweimal deklariert
→ Compilererror
- Abhilfe
  - Definieren einer Konstante, sobald ein Headerfile zum ersten Mal included wird
  - Jedesmal beim include testen, ob die Headerdatei bereits included wurde (die Konstante schon gesetzt ist)

```
/* Modul main.c */
struct Person {
char name[20];
  unsigned int alter;
};
struct Person {
char name[20];
  unsigned int alter;
};
struct List {
  struct Person inhalt;
  struct List *next;
};

void insert (struct List l,
             struct Person p);
void remove (struct List l,
             struct Person p);

int main(void) {
  struct List l;
  struct Person p;
  while (1) {
  }
}
```

- Mit folgendem Konstrukt in jeder Headerdatei wird jede Headerdatei maximal einmal included

```
#ifndef HEADERDATEI_KENNUNG
#define HEADERDATEI_KENNUNG
/* code */
#endif
```

```
/* Modul person.h */
#ifndef PERSON_H
#define PERSON_H
struct Person {
  char name[20];
  unsigned int
  alter;
};
#endif
```

```
/* Modul list.h */
#ifndef LIST_H
#define LIST_H
#include "person.h"

struct List {
  struct Person inhalt;
  struct List *next;
};

void insert (struct List l, struct Person p);
void remove (struct List l, struct Person p);
#endif
```

16.6 make Utility

- Jedes grössere Programm setzt sich aus einer Reihe von Dateien (Module) zusammen
- Schon bei Veränderungen einzelner Dateien kann es erforderlich sein, mehrere Module neu zu kompilieren
- Wird z.B. eine Headerdatei verändert, so sollten alle Quellcodedateien, die diese Headerdatei einschliessen, neu kompiliert werden
- Bei grossen Projekten wird dies sehr schnell unübersichtlich
  → make Utitliy
- make stammt ursprünglich von Unix und hilft, ein Programm aus mehreren Modulen unter Berücksichtigung der Abhängigkeiten zu erstellen
- Beim Aufruf von make werden die Regeln des im aktuellen Verzeichnis liegenden Makefile rekursiv abgearbeitet

16.6.1 Makefile

- Das Makefile enthält Regeln, wann, was und wie etwas auszuführen ist
- Es besteht primär aus
  - Kommentaren; beginnend mit #
  - Variablendefinitionen
  - Expliziten Regeln
- Eine Regel ist wie folgt aufgebaut

```
target: dependencies
<TAB>   command
```

- target: was zu erstellen ist
- dependencies: geben an, wovon target abhängig ist
- command: das oder die abzuarbeitenden Kommandos, die immer dann abgearbeitet werden, wenn eine der dependencies ein jüngeres Modifikationsdatum hat als target
- Der <TAB> vor einem command ist notwendig (keine zusätzlichen Leerzeichen!)
- Die Abarbeitung der Regeln erfolgt rekursiv
- Beispiel: Programm rechner aus Modulen: main.c, add.c, sub.c, mul.c und div.c
- Alle Quellcodedateien nutzen die gemeinsame Headerdatei def.h
- Entsprechendes Makefile (Aufruf: make [all], make rechner oder make clean)

```
# rechner wird aus allen Objektdateien (.o) zusammengebaut; clean dient
# dazu, alle Objektdateien zu entfernen
all: rechner
rechner: add.o sub.o mul.o div.o main.o
      gcc -o rechner add.o sub.o mul.o div.o main.o
clean:
      rm -f *.o rechner

# so entstehen die Objektdateien (def.h und Makefile werden angegeben, um
# auch bei Veränderungen dieser Dateien die Kompilierung neu zu starten)
add.o: add.c def.h Makefile
      gcc -c add.c
sub.o: sub.c def.h Makefile
      gcc -c sub.c
mul.o: mul.c def.h Makefile
      gcc -c mul.c
div.o: div.c def.h Makefile
      gcc -c div.c
main.o: main.c def.h Makefile
      gcc -c main.c
```

- Führt man make all aus, so geschieht folgendes
  - Der Einstiegspunkt des Makefiles ist das target all, dort steht, all hängt von rechner ab
  - Man springt zum target rechner, rechner hängt von add.c, def.h, Makefile ab. Ist eines davon jünger als add.o, so wird add.o gemäss gcc -c add.c kompiliert
  - Dasselbe geschieht mit sub.o, mul.o, etc.
  - Man kehrt wieder zum target rechner zurück. Wurde eine der add.o, sub.o etc. neu kompiliert, so wird gcc -o rechner add.o … ausgeführt
  - Man kehrt zu all zurück, dort gibt es kein command, womit das Makefile verlassen und make beendet wird
- Wird make ohne Parameter ausgeführt, so wird der erste Eintrag im Makefile verwendet. Im obigen Beispielt ensspricht make also make all. Weil das Default-Verhalten von make (ohne Parameter) make all entsprechen sollte, fügt man den all-Eintrag deshalb typischerweise ganz zuoberst ein.

- Im Prinzip wäre hier kein target all nötig, denn mit make rechner erreicht man ja genau dasselbe (all verweist in diesem Beispiel ja einfach auf ein einziges weiteres target (hier: dependendy rechner)). Bei grösseren Projekten stehen hinter dem target all aber oft mehrere Teilprogramme, deshalb macht das target all dort Sinn und man findet es deshalb aus "Konsistenzgründen" meist auch bei kleineren Projekten, damit ganz Allgemein mit make all in jedem Projekt "alles" aktualisiert wird.

16.7 Zusammenfassung

- Um aus Quellcode ein ausführbares Programm zu generieren, werden nacheinander Präprozessor, Compiler und Linker ausgeführt
  - Der Präprozessor macht textuelle Ergänzungen im Quellcode
  - Der Compiler wandelt dann den Quellcode in Objektcode um
  - Der Linker verbindet offene Aufrufe und generiert ein ausführbares Programm
- Bei der modularen Programmierung wird der gesamte Quellcode sinnvoll auf mehrere Dateien verteilt; Funktionsdeklarationen, Konstanten (#define) und Deklarationen von Strukturen gehören in Headerdateien
- Die Standard Library stellt diverse Funktionen für ganz verschiedene Aufgaben zur Verfügung; die Standard Library enthält nichts anderes als Objektcode (z.B. stdio.o); dazu gibt es die entsprechenden Standard Headers (z.B. stdio.h)
- Das make Utility erlaubt es, ein Programm, welches aus mehreren Modulen besteht, effizient zu generieren