ANSI-C Abgeleitete Typen: Zeiger (4)

Verwendung von Zeigern z.B. bei dynamischer Spreicherverwaltung:

- die Funktion **malloc** reserviert Speicher für Werte eines Typs und liefert die Adresse des Speicherbereichs:

```
Typ *Zeigername = (Typ*) malloc(sizeof (Typ));
if (Zeigername == NULL)
{
    ... /* Fehlerbehandlung */
}
```

Anzahl benötigte Bytes

malloc hat Rückgabetyp `void*`

malloc liefert die ungültige Adresse 0 (in ANSI-C als NULL geschrieben), wenn die angeforderte Menge Speicher nicht verfügbar ist.

Achtung: malloc reserviert nur Speicher, initialisiert ihn aber nicht

- mit der Funktion **free** kann (und sollte!) per malloc reservierter Speicher irgendwann wieder freigegeben werden:

```
free (Zeigername);
```

`#include <stdlib.h>` erforderlich, damit malloc und free bekannt sind

ANSI-C Abgeleitete Typen: Felder (3)

Felder und dynamischer Spreicherverwaltung:

- die Funktion **calloc** reserviert Speicher für ein Feld von Werten eines Typs und liefert die Adresse des Speicherbereichs:

```
Typ *Zeigername = (Typ*) calloc(Feldgröße, sizeof (Typ));
if (Zeigername == NULL)
{
    ... /* Fehlerbehandlung */
}
```

- calloc initialisiert den reservierten Speicher mit 0

wird die Initialisierung nicht gebraucht, kann malloc verwendet werden:
`Typ *Zeigername = (Typ*) malloc(Feldgröße * sizeof (Typ));`

- Speicher auch bei calloc mit free wieder freigegeben:

```
free (Zeigername);
```

- Quellcode:

```
#include <stdio.h>
#include <stdlib.h> /* calloc, malloc, free, ... */
#include <stddef.h> /* NULL, size_t, ... */
int main(void)
{
    const int n = 4;
    int i;
    int *a = (int*) calloc((size_t) n, sizeof (int));
    if (a == NULL)
    {
        printf("Speicherreservierung fehlgeschlagen!\n");
        return 1;
    }
```

oder ohne Initialisierung mit 0:
`int *a = (int*) malloc(n * sizeof (int));`

Beispielprogramm Verzweigung

```
#include <stdio.h>
int main(void)
{
    int m, n;
    printf("Zwei Zahlen eingeben: ");
    if (scanf("%d%d", &m, &n) < 2)
    {
        fprintf(stderr, "Eingabefehler !\n");
    }
    else if (m > n)
    {
        printf("Maximum: %d\n", m);
    }
    else
    {
        printf("Maximum: %d\n", n);
    }
    return 0;
}
```

Modell	char	short	int	long	Long long	void*
ILP32	1	2	4	4	8	4
LP64	1	2	4	8	8	8

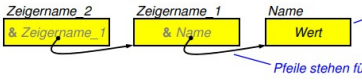
ANSI-C Abgeleitete Typen: Zeiger (1)

Zu jedem Typ kann ein Zeigertyp *(Pointertyp)* abgeleitet werden, indem man in der Variablen-Definition einen Stern `*` vor den Variablen-Namen schreibt.

- **Variablen-Definition**: `Typ Name = Wert;`

```
Typ *Zeigername_1 = &Name;
Typ **Zeigername_2 = &Zeigername_1;
```

- **Wert**:
  Die Adresse eines Speicherbereichs *(Wert 0 bedeutet, der Zeiger zeigt nirgendwohin)*
- **Platzbedarf** je nach Rechner bzw. Compiler:
  `sizeof (int) ≤ sizeof (Typ*)` typisch: 8 Byte
- Grafische Darstellung:



Kästchen stehen für Speicherbereiche

Pfeile stehen für Adresswerte

ANSI-C Abgeleitete Typen: Zeiger (2)

- Zeiger auf konstanten Wert:

```
const Typ Name = Wert;
Typ *Zeigername = &Name; /* Fehler */
const Typ *Zeigername = &Name;
```

Der Wert einer Konstanten kann auch auf dem Umweg über Zeiger nicht geändert werden.

- konstanter Zeiger:

```
Typ Name = Wert;
Typ * const Zeigername = &Name;
```

Ein konstanter Zeiger zeigt während des ganzen Programmlaufs auf denselben Speicherbereich.

- konstanter Zeiger auf konstanten Wert:

```
const Typ * const Zeigername = &Name;
```

- **Inhaltsoperator** `*` macht vom Zeiger adressierten Speicherbereich zugreifbar:

`*Zeigername` *Achtung: Programm-Absturz, wenn der Zeiger den Wert 0 hat*

Inhaltsoperator ist Gegenstück zum Adressoperator:

`*&Name` ist das gleiche wie *Name*

Beispiel-Programm Feld-Zeiger (1)

- Quellcode:

```
#include <stdio.h>
#include <stdlib.h> /* calloc, malloc, free, ... */
#include <stddef.h> /* NULL, size_t, ... */
int main(void)
{
    const int n = 4;
    int i;
    int *a = (int*) calloc((size_t) n, sizeof (int));
    if (a == NULL)
    {
        printf("Speicherreservierung fehlgeschlagen!\n");
        return 1;
    }
```

oder ohne Initialisierung mit 0:
`int *a = (int*) malloc(n * sizeof (int));`

ANSI-C Abgeleitete Typen: String (3)

- Manipulation von C-Strings mit Bibliotheks-Funktionen:

```
char *strcpy(char *s1, const char *s2);
```
kopiert den String s2 in den Speicherbereich s1 und liefert s1 als Rückgabewert

```
char *strcat(char *s1, const char *s2);
```
hängt den String s2 an den String s1 an und liefert s1 als Rückgabewert

```
int strcmp(const char *s1, const char *s2 );
```
Vergleicht die Strings s1 und s2 und liefert 0, wenn die Strings gleich sind, eine Zahl größer 0 bei s1 > s2 bzw. eine Zahl kleiner 0 bei s1 < s2

```
size_t strlen(const char *s);
```
liefert die Länge des Strings s ohne '\0'

```
... /* noch einige weitere str-Funktionen */
```

ANSI-C Benutzerdefinierte Typen: struct (1)

Eine Struktur fasst Werte beliebiger Typen zusammen.

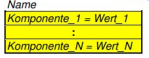
- **Typ-Deklaration**:

```
struct Strukturname
{
    Typ_1 Komponente_1;
    ...
    Typ_N Komponente_N;
};
```

- **Variablen-Definition**: `struct Strukturname Name = {Wert_1, ..., Wert_N};`
- **Wert**: Folge der Komponenten-Werte.
- **Platzbedarf**: $\sum_{i=1}^{N}$ `sizeof (Typ_i)` ≤ `sizeof(struct Strukturname)`

wegen Alignment der Komponenten

- Grafische Darstellung:



ANSI-C Funktionen: main

Jedes Programm muss genau eine Funktion mit dem Namen `main` enthalten.

- es gibt eine Variante mit und eine ohne Parameter:

```
int main(int argc, char *argv[])
{
    ...
}
```

```
int main(void)
{
    ...
}
```

Erklärung der Parameter:

argv	*Feld von String-Pointern (argument vector).*
argc + 1	*Feldgröße (argument count).*
argv[0]	*Programm-Name (Kommando)*
argv[1]	*erstes Kommandozeilen-Argument*
argv[argc – 1]	*letztes Kommandozeilen-Argument*
argv[argc]	*0 (NULL-Pointer)*

ANSI-C Benutzerdefinierte Typen: struct (2)

- **Komponentenauswahl-Operatoren** (Punkt und Pfeil):

```
Name.Komponente_1
Zeigername->Komponente_1
```

Pfeil ist Kurzschreibweise für `(*Zeigername).Komponente_1`

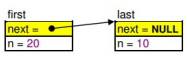
- Adresse einer Komponente:

```
&Name.Komponente_1
&Zeigername->Komponente_1
```

Adresse der ersten Komponente ist Adresse der Struktur insgesamt

- **Verkettete Strukturen** enthalten einen Zeiger auf den eigenen Strukturtyp:

```
struct int_list
{
    struct int_list *next; /* Verkettung */
    int n;
};
struct int_list last = {NULL, 10};
struct int_list first = {&last, 20};
```



ANSI-C Übersetzungseinheiten: Vergleich mit Java (2)

Beispielklasse in Java und entsprechende Übersetzungseinheit in ANSI-C:

```
// IntegerMethods.java
public final class IntegerMethods {
    private IntegerMethods() { }

    public static
    int max(final int m, final int n) {
        return m > n ? m : n;
    }

    public static
    int min(final int m, final int n) {
        return m < n ? m : n;
    }
}
```

```
/* integer_methods.h */
#ifndef INTEGER_METHODS_H
#define INTEGER_METHODS_H
int int_max(int, int);
int int_min(int, int);
#endif
```

```
/* integer_methods.c */
#include "integer_methods.h"
int int_max(int m, int n) {
    return m > n ? m : n;
}
int int_min(int m, int n) {
    return m < n ? m : n;
}
```

ANSI-C Standard-Bibliothek: Überblick

Die Schnittstelle der Standard-Bibliothek ist (teilweise etwas beliebig) in diverse Header-Dateien aufgeteilt:

- Grundlegendes zur Sprachunterstützung *(NULL, size_t, malloc, calloc, free, ...)*
  `<stdarg.h>` `<stddef.h>` `<stdlib.h>`
- Ein-/Ausgabe
  `<stdio.h>`
- Umgang mit Zeichen und Zeichenketten
  `<ctype.h>` `<string.h>`
- Umgang mit Zahlen:
  `<float.h>` `<limits.h>` `<math.h>` `<locale.h>`
- Umgang mit Datum und Zeit
  `<time.h>`
- Umgang mit Fehlern und Ausnahmesituationen
  `<assert.h>` `<errno.h>` `<setjmp.h>` `<signal.h>`

Makefile: C-Beispiel hallo (1)

```
# Makefile
# Kommando-Variablen
CC = gcc
CFLAGS = -W -Wall -ansi -pedantic
CPPFLAGS = -I.
RM = rm -f

# Hilfsvariablen
TARGET = hallo
OBJECTS = gruss.o
SOURCES = $(TARGET).c $(OBJECTS:.o=.c)
HEADERS = $(OBJECTS:.o=.h)

# Musterregeln
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
...
```

Variablen für die vordefinierten C-Übersetzungsregeln

Include-Dateien im aktuellen Verzeichnis suchen

die C-Übersetzungsregel ist vordefiniert und braucht deshalb nicht angegeben zu werden

Makefile: C-Beispiel hallo (2)

```
...
# Standardziele
.PHONY: all clean
all: $(TARGET)
clean:
    $(RM) $(TARGET) $(TARGET).o $(OBJECTS) depend

depend: $(SOURCES) $(HEADERS)
    $(CC) $(CPPFLAGS) -MM $(SOURCES) > $@

# Ziele zur Programmerstellung
$(TARGET): $(TARGET).o $(OBJECTS)
    $(CC) $(LDFLAGS) $^ -o $@

# Abhaengigkeiten
include depend
```

Pseudoziele

Beispiel-Programm: std::string

```
#include <iostream>
#include <string>

int main()
{
    std::string a = "halli"; // a("halli")
    std::string s = "hallo"; // s("hallo")
    std::string t; // leerer String

    // compare, copy and concatenate strings
    if (a < s) // operator<(a, s)
    {
        t = a + s; // t.operator=(operator+(a, s))
    }

    // print string values and addresses
    std::cout << a << '\n' << s << '\n' << t << '\n'; // operator<<(..., ...)
    std::cout << sizeof a << '\n' << sizeof s << '\n' << sizeof t << '\n';
    std::cout << a.length() << '\n' << s.length() << '\n' << t.length() << '\n';
}
```

make: Beispiel (1)

- einfacher Bauplan für das Programm hallo:

```
# Makefile
hallo: hallo.o gruss.o
    gcc hallo.o gruss.o -o hallo
hallo.o: hallo.c gruss.h
    gcc -c hallo.c
gruss.o: gruss.c gruss.h
    gcc -c gruss.c
```

Abhängigkeit (hallo abhängig von zwei Objektdateien)

Kommando (erzeugt hallo aus zwei Objektdateien)

Tabulator vor dem Kommando nicht vergessen

- Aufruf zum Erstellen bzw. Aktualisieren des Programms:

```
make -f Makefile hallo
make   # tut das gleiche, weil Makefile Standardname und hallo erstes Ziel ist
```

Makefile: implizite Regeln (2)

make hat für die wichtigsten Dateitypen **vordefinierte implizite Regeln**, z.B.:

- Übersetzen und binden eines C-Programms mit nur einer Quelldatei:

```
.c:
    $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o $@ $<
```

Zielendung ist leer (ausführbare Programme unter Linux ohne Endung)

`make hallo` erzeugt aus `hallo.c` das ausführbare Programm `hallo`

- Übersetzen einer C-Quelle:

```
.c.o:
    $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

automatische Variablen für Ziel und Quelle

`make hallo.o` erzeugt aus `hallo.c` die Objektdatei `hallo.o`

- Kommandos in vordefinierten Regeln sind mit Variablen definiert, um sie leicht an unterschiedliche Plattformen anpassen zu können:

```
CC=gcc
CPPFLAGS=
CFLAGS=
LDFLAGS=
```

Präprozessor-Optionen (hier leer)

Compiler-Optionen (hier leer)

Linker-Optionen (hier leer)

Einige wichtige Unterschiede:

- in C++ können Klassen mehrere Oberklassen haben *(Mehrfachvererbung)*
- in C++ sind Klassen als Werttyp verwendbar (sind sogar vorrangig so gedacht)
  deshalb Objekte nicht nur auf dem Heap, sondern auch auf dem Stack und auch ineinander verschachtelte möglich
  deshalb Copy-Konstruktor und Zuweisungsoperator in jeder Klasse
- in C++ Operator-Overloading möglich
  Operatoren können dadurch auf benutzerdefinierte Typen angewendet werden
- in C++ kein Garbage-Collector
  deshalb Operator `delete` zur Speicherfreigabe und in jeder Klasse ein Destruktor und in neueren Versionen Bibliotheksklassen zur Kapselung von Zeigern (intelligente Zeiger)
- in C++ bevorzugt generische Programmierung mit Templates
  Templates bieten wesentlich mehr Möglichkeiten als die Generics von Java

Syssprog Probekklausur Lösungen

Aufgabe 1:
a.)

```
int a;                          a = 1;
int *b;                         b = &a;
double c[2];                    c[0] = 2.3;
struct Raum d;                  c[1] = 4.5;
struct Raum *e;

struct Raum                     d.zimmer = 109;
{                               d.gebauede = ' C ';
int zimmer;                     d.art = "Hoersaal";
char gebauede;
const char *art;                e = (struct Raum*) malloc (sizeof(struct Raum));
}d;                             e -> zimmer = 124;
                                e -> gebauede = ' F ';
                                e -> art = "Buero";
```

b.)

ILP 32: 4 + 4 + 1 = 9 Byte ohne Ausrichtung
4 + 4 + 4 = 12 Byte mit Ausrichtung

LP 64: 8 + 4 + 1 = 13 Byte ohne Ausrichtung
8 + 8 + 8 = 16 Byte mit Ausrichtung

c.)

```
/*wuerfeltest.c */
#include <stdio.h>
#include <stdlib.h>
#include "wuerfel.h"

int main(int argc, char* argv[])
{
double k = atof(argv[1]);
double f = Wuerfel.oberflaeche(k);
double v = Wuerfel.volumen(k);

printf("Kantenlaenge %f, Oberflaeche %f, Volumen %f%n", k, f, v);
return 0;
```

d.)
Es wird eine Null übergeben deshalb bricht es auch mit einer NullPointerException ab. Bei argc und argv ist kein Wert vorhanden deshalb wird eine Null übergeben. Mit atof wird versucht ein Null-Wert in einen Int-Wert umzuwandeln. Was nicht funktionieren kann.

Aufgabe 2:

a.)

```
/* quadrat.h */

#ifndef   QUADRAT_H
#define   QUADRAT_H

double quadrat_flaeche(double);

#endif
```

```
/* quadrat.c */
#include "quadrat.h"

static double zumquadrat(double);
double quadrat_flaeche(double s)
{
return zumquadrat(s);
}

static double zumquadrat(double d)
{
return d * d;
}
```

b.)

```
/* wuerfel.h */
#ifndef   WUERFEL_H
#define   WUERFEL_H

double wuerfel_oberflaeche(double);
double wuerfel_volumen(double);

#endif
```

```
/* wuerfel.c */
#include "wuerfel.h"
#include "quadrat.h"

double wuerfel_oberflaeche(double k)
{
return Quadrat.flaeche(kantenlaenge) * 6;
}

double wuerfel_volumen(double v)
{
return Quadrat.flaeche(kantenlaenge) * kantenlaenge;
}
```

Aufgabe 3:

a.)

```
size_t strlen(const char *s)
{
size_t size = 0;
int i;
for(i = 0; s[i]!='\0'; ++i)
{
++ size;
}
return size;
}
```

Aufgabe 4:

```
PDFLATEX = pdflatex
TARGET = hallo.pdf

%.pdf : %.tex
|-> ${PDFLATEX}$<
.PHONY = all clean
all = $(TARGET)

clean =
|-> $(RM) $(TARGET) $(TARGET : .pdf = .aux) $(TARGET : .pdf = .log)
```

b.)

```
const char *s = "Halli";
const char *t = "Hallo";

char *st = (char*) malloc (sizeof(strlen(s) + strlen(t) + 1);

strcpy(st, s);
strcat(st, t);
```

Aufgabe 5:

a.)
Absturz, Endlosschleife, falsche Ergebnisse, Speicherüberlauf

b.)
Absturz: ddd valgrind
Endlosschleife: ddd -> Deadpoint setzen
Speicherüberlauf: valgrind
Fehlverhalten: Hypothese bilden -> beweisen , -> widerlegen

Aufgabe 6:

Rückgabewert -1
errno

Aufgabe 7:

a.)
boolean, Namespaces, überladen von Methoden, Operatoren überladen, Kapselung

b.)
Mehrfachvererbung in C++, GarbageCollector in Java
Alles kann in C++ geworfen werden

```
struct element_prototype
{
    // Eigentlicher Inhalt (hier: int):
    int value;

    // Zeiger auf das vorheriges und nächste Element:
    element_prototype * prev;
    element_prototype * next;
};

typedef element_prototype element_type;
```

```
// Zeiger auf Elemente deklarieren:
element_type *e0, *e1;

int init_list()
{
    // Dynamischen Speicherplatz für Elemente reservieren:
    e0 = (element_type *) malloc(sizeof *e0);
    e1 = (element_type *) malloc(sizeof *e1);

    // Fehlerkontrolle:
    if (e0 == NULL) || (e1 == NULL)
        return 1;

    // Referenzen anpassen:
    e0->prev = NULL;
    e0->next = e1;

    e1->prev = e0;
    e1->next = NULL;

    // Normaler Rückgabewert:
    return 0;
}
```

Code-Beispiele

```
struct spieler
{
char name;
char vorname;
int alter;
struct spieler *next;
struct spieler *previous;
};

struct spieler *previous; //vorgänger
struct spieler *next;     // nachfolger
```

```
// Zeiger auf Elemente deklarieren:
element_type *e0, *e1;

int init_list()
{
    // Dynamischen Speicherplatz für Elemente reservieren:
    e0 = (element_type *) malloc(sizeof *e0);
    e1 = (element_type *) malloc(sizeof *e1);

    // Fehlerkontrolle:
    if (e0 == NULL) || (e1 == NULL)
        return 1;

    // Referenzen anpassen:
    e0->next = e1;
    e1->next = NULL;

    // Normaler Rückgabewert:
    return 0;
}
```

```
/*
 * stringvar.c
 *
 * Beispiel-Anwendung von str-Funktionen
 *
 * Autor: H.Drachenfels
 * Erstellt am: 25.2.2015
 */

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h> /* strxxx functions */

int main(void)
{
    char a[] = "halli";
    const char *s = "hallo";
    char *t = NULL;

    /*----------------------------- compare, copy and concatenate strings */
    if (strcmp(a, s) < 0)
    {
        t = (char*) malloc(sizeof a + strlen(s));
        if (t == NULL)
        {
            fprintf(stderr, "Kein Heap-Speicher mehr\n");
            return 1;
        }

        strcat(strcpy(t, a), s); /* or: strcpy(t, a); strcat(t, s); */
    }
    else
    {
        t = (char*) calloc(1, sizeof (char));
        if (t == NULL)
        {
            fprintf(stderr, "Kein Heap-Speicher mehr\n");
            return 1;
        }
    }

    /*-------------------------------- print string values and addresses */
    printf("a = %p %s\ns = %p %s\nt = %p %s\n",
           (void*) a, a, (void*) s, s, (void*) t, t);

    printf("sizeof a = %lu\n", (unsigned long) sizeof a);
    printf("sizeof s = %lu\n", (unsigned long) sizeof s);
    printf("sizeof t = %lu\n", (unsigned long) sizeof t);

    printf("strlen(a) = %lu\n", (unsigned long) strlen(a));
    printf("strlen(s) = %lu\n", (unsigned long) strlen(s));
    printf("strlen(t) = %lu\n", (unsigned long) strlen(t));

    s = a; /* copies the address, not the string */
    /* a = s; syntax error */

    free(t);

    return 0;
}
```