Übersetzung des Programms:

Slide 57

```
gcc -c -o nachfolger.o nachfolger.c
gcc -c -o main.o main.c
gcc -o main main.o nachfolger.o
```

14 Fortgeschrittene Programmierung

14.1 Programmieren mit Zeigern

Zeiger-Arithmetik

Slide 58

```
long *zeiger, feld[20];
zeiger = feld;
```

Feld und Zeiger können nun äquivalent verwendet werden: `feld[1]` entspricht `zeiger[1]`.

Zeiger können in Einheiten der Größe ihres Typs inkrementiert werden:

```
zeiger++;
```

Zeiger zeigt nun auf `feld[1]` (d.h. `zeiger == &feld[1]`).

Feld-Addressierung	Zeiger-Adressierung
feld[0]	*feld
&feld[1]	feld + 1
feld[1]	*(feld + 1)

Iterieren mit Zeigern benötigt keine Schleifenvariable:

Slide 59

```
int main(int argc, char *argv[], char *envp[])
{
  while (*envp)
    {
      printf("%s\n", *envp++);
    }
  return 0;
}
```

14.2 Dynamische Speicherverwaltung

Vier Bibliotheksfunktionen verwalten den Heap:

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

Slide 60

`void *malloc(size_t size)`

fordert `size` Bytes Hauptspeicher an.

`void *calloc(size_t nmemb, size_t size)`

fordert null-initialisierten Speicher für `nmemb` Feldelemente der Größe `size` an.

`void free(void *ptr)`

gibt mit `malloc()` oder `calloc()` angeforderten Speicherblock an Adresse `ptr` wieder frei.

Slide 61

`void *realloc(void *ptr, size_t size)`

verändert die Größe des mit `malloc()` oder `calloc()` angeforderten Speicherblocks an Adresse `ptr` auf `size` Bytes und liefert die Adresse des neuen (evtl. verschobenen) Blocks zurück. Es ist sowohl ein Verkleinern als auch ein Vergrößern des Speicherblocks möglich.

Slide 62

```
#include <stdlib.h>

struct elem
{
  char name[100];
  int zensur;
  struct elem *next, *prev;
};
typedef struct elem *elem_p;

struct list
{
  elem_p first, last;
};
typedef struct list *list_p;

void list_init(list_p l)
{
  l->first = l->last = NULL;
}
```

```
void list_insert(list_p l, elem_p e)
{
  e->next = l->first;
  e->prev = NULL;

  if (l->first) l->first->prev = e;
  else l->last = e;

  l->first = e;
}

void list_delete(list_p l, elem_p e)
{
  if (e->prev) e->prev->next = e->next;
  else l->first = e->next;

  if (e->next) e->next->prev = e->prev;
  else l->last = e->prev;
}
```

```
elem_p schueler;
list_p klasse9a;

klasse9a = (list_p) malloc(sizeof(struct list));
list_init(klasse9a);
```

Slide 63

```
schueler = (elem_p) malloc(sizeof(struct elem));
strcpy(schueler->name, "Heinz");
schueler->note = 4;

list_insert(klasse9a, schueler);
```

14.3 Unix-Systemrufe

Zugriff auf Unix-Kernobjekte:

`#include <unistd.h>`

Anhaltspunkte:

Slide 64

- Dateien und Verzeichnisse: `open()`, `opendir()`, `close()`, `closedir()`, `read()`, `write()`, `lseek()`, `unlink()`, `rmdir()` („`man 2 open`“ usw.)
- Pipes: `pipe()`
- Signale: `kill()`, `signal()`, `alarm()`, `pause()`
- Nutzer und Gruppen: `chown()`, `fchown()`, `getuid()`, `getgid()`

Slide 65

- Aktuelles Verzeichnis: `chdir()`, `fchdir()`, `getcwd()`
- Dateideskriptoren: `dup()`, `dup2()`, `select()`
- Prozesse: `fork()`, `execve()`, `wait()`, `waitpid()`

Beispiel

Slide 66

```
pid_t pid;
int status;

if ((pid = fork()) == 0)
  {
    /* Kindprozeß */
    execl("/bin/sh", "sh", "-c", "ls -l | more");
    /* kehrt nie zurück */
  }
/* Elternprozeß */
if (waitpid(pid, &status, 0) == -1)
  printf("Error!\n");
else
  printf("Subprozeß-Rückgabewert: %d\n", WEXITSTATUS(status));
```

14.4 Objektorientiertes Programmieren

Ausgehend vom OO-Paradigma: Objekt = Daten + Methoden

Slide 67

```
struct figur {
  int x, y;

  int (*draw)(struct figur *f);
}
```

Slide 68

```
struct figur *new_circle(void)
{
  struct figur *f = malloc(sizeof(struct figur));
  if (f)
    {
      f->draw = circle_draw;
      f->x = f->y = 0;
    }
  return f;
}

struct figur *k;
k = new_circle();
k->x = 15;  k->y = 76;
k->draw(k);               /* entspricht (*k->draw)(k); */
```