C-Auffrischungskurs

Prof. Dr. Christian Forler

WiSe 2019/20

Inhaltsverzeichnis

Disclaimer

Disclaimer

Kapitel 5 wurde durch Handouts von Prof. Dr. Schimkat inspiriert.

Section 1

Call by Values vs. Call by Address

Call by Values vs. Call by Address

```
1  // Call by value
2  void swap1(int a, int b) {
3    int t = a;
4    a = b;
5    b = t;
6  }
7
8  // Call by address
9  void swap2(int *a, int *b) {
10   int t = *a;
11   *a = *b;
12   *b = t;
13 }
```

Call by Value

```
20 int main() {
21   int x= 1, y= 2;
22
23   printf("%d␣%d\n", x ,y);
24   swap1(x ,y);
25   printf("%d␣%d\n", x ,y);
26 }
```

- Bei dem Aufruf von `swap1()` (Zeile 24) wird der lokalen Variable `a` den Wert `1` und `b` den Wert `2` zugewiesen.
- In den Zeilen 3-5 werden die Werte von `a` und `b` vertauscht.
- Am Ende von `swap1()` (Zeile 6) werden die lokalen Variablen `a`, `b` und `t` *gelöscht*.

$\Longrightarrow$ Die Vertauschung von `a` und `b` hat **keinen** Einfluss auf die Werte von `x` und `y`.

Call by Address/Reference

```
30  int main() {
31    int x= 1, y= 2;
32
33    printf("%d␣%d\n", x ,y);
34    swap2(&x ,&y);
35    printf("%d␣%d\n", x ,y);
36  }
```

- Bei dem Aufruf von `swap2()` (Zeile 34) wird der lokalen Variable `a` die Adresse von `x` und `b` die Adresse von `y` zugewiesen.
- In den Zeilen 10-12 werden die Inhalte der Speicherzellen auf die `a` und `b` zeigen getauscht.
- Am Ende von `swap2()` (Zeile 12) werden die lokalen Zeiger `a`, `b` und `t` *gelöscht*.

$\Longrightarrow$ Die Vertauschung von `a` und `b` hat Einfluss auf `x` und `y`.

Section 2

Aufzählungs- und Verbundstyp (enum und structs)

Aufzählungstyp (enum)

Syntax

```
enum identifier { enumerator-list }
```

- Enumeratoren sind Bezeichner welche aus Großbuchstaben und Unterstrichen bestehen (Beispiel: `MONDAY`)
- Bei der Deklaration können Enumeratoren Ganzzahlenwerte zugewiesen werden (Beispiel: `MAY=5`).

Beispiele

```
enum priorities { LOW, MEDIUM, HIGH };

typedef enum { ONE=1, TWO=2, THREE=3 , TEN=10} numbers_t;
```

Enums sind Konstanten

- Aufzählungstypen werden in C als Integer realisiert.
- Bei **enum** Werten handelt es sich um Konstanten.
- Die folgenden zwei Codefragmente sind semantisch äquivalent.
- Aufzählungstypen erhöhen die Lesbarkeit von Code.

```
enum priorities { LOW, MEDIUM, HIGH };

void send_packet(enum priorities p);
```

```
const unsigned int  LOW=0;
const unsigned int  MEDIUM=1;
const unsigned int  HIGH=2;

void send_packet(int p);
```

Beispiel

```
1  #include <stdio.h>
2
3  typedef enum priorities { LOW, MEDIUM, HIGH } prio_t;
4
5  void do_something(prio_t p) {
6    switch(p) {
7    case HIGH:   puts("Call␣for␣reinforcement.");
8    case MEDIUM: puts("Send␣guards.");
9    case LOW:    puts("Activate␣lights.");
10   }
11   puts("");
12 }
13
14 int main() {
15   prio_t p1 = HIGH;
16   prio_t p2 = 0;
17
18   do_something(p2);
19   do_something(p1);
20 }
```

Der Verbundstyp (struct)

Syntax

```
struct identifer {
  type member1;
  type member2;
  ...
  type memberN;
}
```

- Ein `struct` besteht aus mehreren Datentypen (*Member*).
- Ein Member kann wiederum ein `struct` sein.

Beispiele

```
struct point {
  int x;
  int y;
};
```

Alignment

```
struct foo {
  char a;
  int b;
};
```

- Das Alignment eines `structs` hängt von dem Compiler ab.
- Die Größe des obigen Structs ist daher `undefiniert`.
- Falls das `struct` als Maske dienen soll muss es richtig ausgerichted (aligned) werden.
- Das Pragma `pack()` legt das Byte-Alignment fest.
- Das Pragma `pack(push)` sichert das Alignment.
- Das Pragma `pack(pop)` stellt das Alignment wieder her.

Beispiel

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct foo {
5    char a;
6    int b;
7  };
8
9  #pragma pack(push)
10 #pragma pack(1)
11 struct bar {
12   char a;
13   int b;
14 };
15 #pragma pack(pop)
16
17 int main() {
18   char *test = "ABCDE";
19   struct foo f;
20   struct bar b;
21
22   printf("size␣foo:␣%lu\n␣", sizeof(struct foo));
23   printf("size␣bar:␣%lu\n␣", sizeof(struct bar));
24
25   memcpy(&f, test, 5);
26   memcpy(&b, test, 5);
27
28   printf("foo:␣%c␣0x%08x␣\n␣", f.a, f.b);
29   printf("bar:␣%c␣0x%08x␣\n␣", b.a, b.b);
30 }
```

Section 3

Headerdateien und Makefiles

Headerdateien

- Die folgenden 3 Dinge gehören in eine Headerdatei.
  1. Typdefinitionen
  2. Deklaration von globalen Konstanten
  3. Funktionsprototypen
- Headerdateien erhöhen die Lesbarkeit.
- Headerdateien bringen etwas Ordnung in das Chaos.
- Headerdateien *dokumentieren* eine compilierte Datei.

Mehrfacheinbindung

foobar.h

```
enum priority { LOW, MEDIUM, HIGH };

 int   do_something(enum priority *prio);
```

- Die Headerdatei `foobar.h` kann nur von einem Programmteil eingebunden werden.
- Durch die Mehrfacheinbindung würde die Aufzählung `priority` und die Funktion `do_something` mehrfach deklariert werden. Dies ist in C nicht erlaubt.
- **Lösung: Include-Guard.**

Include-Guards

Lösung mit Präprozessor-Makro

```
#ifndef FOOBAR_H
#define FOOBAR_H

enum priority { LOW, MEDIUM, HIGH };

int  do_something(enum priority *prio);

#endif // FOOBAR_H
```

Lösung mit Pragma once

```
#pragma once

enum priority { LOW, MEDIUM, HIGH };

int  do_something(enum priority *prio);
```

Beispiel: point

```c
#pragma once

typedef struct {
  int x;
  int y;
} point;

void p_print(const point *p);
void p_add(const point *a, const point *b, point *result);
void p_subtract(const point *a, const point *b, point *result);
```

```c
#include <stdio.h>
#include "point.h"

void p_print(const point *p) {
  printf("(%d, %d)", p->x, p->y);
}

void p_add(const point *a, const point *b, point *result) {
  result->x  = a->x + b->x;
  result->y  = a->y + b->y;
}

void p_subtract(const point *a, const point *b, point *result) {
  result->x  = a->x - b->x;
  result->y  = a->y - b->y;
}
```

Beispiel: pointdemo

```
1  #include <stdio.h>
2  #include "point.h"
3
4  static void output(const point *a, const point *b,
5                     point *c, const char *op) {
6    p_print(a);
7    printf("␣%c␣", *op);
8    p_print(b);
9    printf("␣=␣");
10   p_print(c);
11   puts("");
12 }
13
14 int main() {
15   point foo = { 2 , 3};
16   point bar = { 4 , 2};
17   point r;
18
19   p_add(&foo, &bar, &r);
20   output(&foo, &bar, &r, "+");
21   p_subtract(&foo, &bar, &r);
22   output(&foo, &bar, &r, "-");
23 }
```

Makefiles

- Mit Hilfe von Makefiles lassen sich Software wie Programme oder Projekte (im folgenden Targets genannt) bauen.

Makefiles

- Mit Hilfe von Makefiles lassen sich Software wie Programme oder Projekte (im folgenden Targets genannt) bauen.
- Für die Erstellung von Objektdateien gibt es eine Default-Rezept.

Makefiles

- Mit Hilfe von Makefiles lassen sich Software wie Programme oder Projekte (im folgenden Targets genannt) bauen.
- Für die Erstellung von Objektdateien gibt es eine Default-Rezept.
- Einfache C-Programmen die nur aus einer C-Datei bestehen lassen sich auch mit einem Default-Rezept bauen.

Makefiles

- Mit Hilfe von Makefiles lassen sich Software wie Programme oder Projekte (im folgenden Targets genannt) bauen.
- Für die Erstellung von Objektdateien gibt es eine Default-Rezept.
- Einfache C-Programmen die nur aus einer C-Datei bestehen lassen sich auch mit einem Default-Rezept bauen.
- Für nicht triviale Programme muss eine Zutatentliste mit angegeben werden.

Makefiles

- Mit Hilfe von Makefiles lassen sich Software wie Programme oder Projekte (im folgenden Targets genannt) bauen.
- Für die Erstellung von Objektdateien gibt es eine Default-Rezept.
- Einfache C-Programmen die nur aus einer C-Datei bestehen lassen sich auch mit einem Default-Rezept bauen.
- Für nicht triviale Programme muss eine Zutatentliste mit angegeben werden.
- Komplexere Programme benötigen ein eigenes Rezept. (siehe Software-Engineering Vorlesung)

Ausgewählte vordefinierte Variablen

- `CC`: C-Compiler
- `CXX`: C++ Compiler
- `CFLAGS`: C-Compiler Flags
- `CFLAGS`: Linker Flags
- `RM`: `rm -f`
- `$@`: Name des aktuellen Targets
- `$^`: aktuelle Zutatenliste
- `$<`: Erste Zutat

Beispiel: Makefile

```
1  WARNFLAGS = -W -Wall -Werror
2  OPTFLAGS = -O3
3  DEBUGFLAGS = -ggdb3 -DDEBUG
4  CFLAGS += $(WARNFLAGS)
5  binaries= enumdemo alignment pointdemo errordemo
6
7  ifdef DEBUG
8    CFLAGS += $(DEBUGFLAGS)
9  else
10   CFLAGS += $(OPTFLAGS)
11 endif
12
13 all: $(binaries)
14
15 pointdemo: point.c
16
17 clean:
18   $(RM)  *~ $(binaries) *.o
```

- Die Programme `enumdemo`, `alignment` und `errordemo` werden mit Hilfe einer Default-Regel gebaut.
- Das Programm `pointdemo` kann nicht mit einer Default-Regel erstellt werden, da es von `point.c` abhängt.

Section 4

Fehlermeldung

Fehlermeldung

- Fehler werden auf der Standardfehlerausgabe (Filedeskriptor 2) und nicht auf der Standardausgabe (Filedeskriptor 1) ausgegeben.
- Dies ermöglich es die Fehlermeldungen in eine Datei zu schreiben (`# cat eee  2> error.log`).
- In `stdio.h` sind die folgenden Konstanten definiert.
  - `extern FILE *stdin;` Standardeingabe (fd 0)
  - `extern FILE *stdout;` Standardausgabe (fd 1)
  - `extern FILE *stderr;` Fehlerausgabe (fd 2)
- In `unistd.h` sind die folgenden Konstanten definiert.
  - `#define STDIN_FILENO 0`
  - `#define STDOUT_FILENO 1`
  - `#define STDERR_FILENO 2`

Eigene Fehlermeldungen ausgeben

- Mittels `fprintf(stderr,...)` lassen sich Fehlermeldungen auf `stderr` ausgeben
- Alternative ist `puts(string, stderr)`
- **Beispiel:** `puts("Fehler", stderr)`
- **Beispiel:**

```
fprintf(stderr, "%d: Invalide Value\n", foo)
```

Errno

- In dem Headerfile `errno.h` ist die Systemvariable `errno` definiert.
- Sämtliche Systemaufrufe und viele Aufrufe der Standard-C Bibliothek (libc) setzten `errno`, falls ein Fehler auftritt.
- `errno == 0`: Es ist noch kein Fehler aufgetreten.
- `errno != 0`: Es ist ein Fehler aufgetreten,
- In POSIX.1-2001 sind die Fehlernummen definiert
  - `ENOENT` Datei oder Verzeichnis nicht vorhanden
  - `EISDIR` Ist ein Verzeichnis
  - `EACCES` Keine Berechtigung
  - ...

Systemfehlermeldungen ausgeben

```
#include <string.h>

char *strerror(int errnum);
```

Die Funktion `strerror()` liefert für einen Fehlercode `errnum` die passende POSIX.1-2001 Fehlermeldung als C-String zurück.

Beispielnutzung: `emsg = strerror(errno);`

```
#include <stdio.h.h>

void perror(const char *s);
```

Die Funktion `perror` gibt den C-String `s` gefolgt von `strerror(errno)` auf der Standardfehlerausgabe (`stderr`) aus. Die beiden C-Strings werden durch `": "` voneinander getrennt.

Beispiel: Fehlermeldungen

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5
6  void usage() {
7    fputs("errordemo␣<file>␣...\n", stderr);
8
9    exit(EXIT_FAILURE);
10 }
11
12 int main(int args, char *argv[]) {
13   if (args < 2) usage();
14
15   for(int i=1; i<args; i++) {
16     int fd = open(argv[i], O_RDONLY);
17     if(fd < 0) perror(argv[i]);
18     else close(fd);
19   }
20 }
```

Section 5

Guter Stil

Namenskonventionen für Bezeichner

- **Funktionen**: Kleinbuchstaben und `'_'`.
  Beispiel: `getline(), event_handler()`
- **Variable**: Kleinbuchstaben und `'_'`.
  Beispiel: `int wave_length; char *text;`
- **Globale Variablen**: Präfix `g_`
  Beispiel: `int g_wave_length; char *g_text;`
- **Konstanten**: Großbuchstaben und `'_'`.
  Beispiel: `const int MAX_LEN, #define LINE_LENGTH 25`
- **Typen**: Präfix `'_t'`.
  Beispiel: `typedef unsigned char uchar_t;`

Rückgabewert

Gegeben

```
#include <stdbool.h>
bool result;
```

Schlechter Code

```
if(result == true) {
  return true;
}
else {
  return false;
}
```

Guter Code

```
return result;
```

Berechnungen

Schlechter Code

```
int bar(int a, int b) {
  int result;
  result = a*a + b + 23;
  return result;
}
```

Guter Code

```
int bar(int a, int b) {
  return a*a + b + 23;
}
```

Parameterübergabe

Schlechter Code

```
void print_demo(void) {
  double x = 2.3;
  printf("%f\n", sqrt(x) );
  x = 4.2;
  printf("%f\n", sqrt(x) );
}
```

Guter Code

```
void print_demo(void) {
  printf("%f\n", sqrt(2.3) );
  printf("%f\n", sqrt(4.2) );

}
```

Berechnung und Ausgabe vermischen

Schlechter Code

```
int foo(int x) {
  int result = (x + x) * x;
  printf("%d\n", x);
  return result;
}
```

Guter Code

```
int foo(int x) {
  return (x + x) * x;
}

void print_foo(int x) {
  printf("%d\n", foo(x));
}
```

Section 6

Häufige Programmierfehler

Nichteinhaltung von Array-Grenzen

Falsch

```
#include <stdio.h>
int main() {
  int a[3] = { 10, 20, 30};

  for(int i=0; i<=3; i++)  printf("%d␣",a[x]);
  puts("");
  return 0;
```

Richtig

```
#include <stdio.h>
int main() {
  int a[3] = { 10, 20, 30};
  for(int i=0; i<3; i++)  printf("%d␣",a[x]);

  puts("");
  return 0;
```

Fehlende Abbruchbedingung bei Rekursion

Falsch

```
inf factorial(int n) {
  return n * factorial(n-1);
}
```

Richtig

```
inf factorial(int n) {
  if(n==1) return 1;
  return n * factorial(n-1);
}
```

Verwendung von falschen Fließkommaliteralen

Falsch

```
#include <stdio.h>
int main() {
  float  a = 0.1;
  if (a > 0.1) puts("Gnarf");
}
```

Richtig

```
#include <stdio.h>
int main() {
  float  a = 0.1;
  if (a > 0.1f) puts("Gnarf");
}
```

Anmerkung

Bei dem Literal `0.1` handelt es sich um ein Double-Literal.
Float-Literale enden mit `f`.

Ein Semikolon zuviel 1/2

Falsch

```
#define MAX 100;

int main() {
  int a[MAX];
}
```

Richtig

```
#define MAX 100

int main() {
  int a[MAX];
}
```

Ein Semikolon zuviel 2/2

Falsch

```
int abs(int a) {
  if(a < 0); {
    a = -a;
  }
  return a;
}
```

Richtig

```
int abs(int a) {
  if(a < 0) {
    a = -a;
  }
  return a;
}
```

Buffer Overflow

Falsch

```
#include <stdio.h>

 int main() {
   char buf[10];
   puts("What␣is␣your␣name?");
   scanf("%s", buf);
 }
```

Richtig

```
#include <stdio.h>

int main() {
  char buf[10];
  puts("What␣is␣your␣name?");
  scanf("%9s", buf);
}
```

Benutzereingabe richtig gut einlesen

```
#include <stdlib.h>
#include <stdio.h>
#include <readline/readline.h>

int main() {
  char *name = readline("What is your name? ");
  printf("Hi %s.\n", name);
  free(name);

  return EXIT_SUCCESS;
}
```

Kompilierung

```
gcc -ggdb3 -W -Wall -Werror -lreadline -o hi hi.c
```

Anmerkung

Um die Funktion `readline()` nutzen zu können muss das Paket `libreadline-dev` installiert sein.