

Systemnahe Programmierung in C

50 Beispiele

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

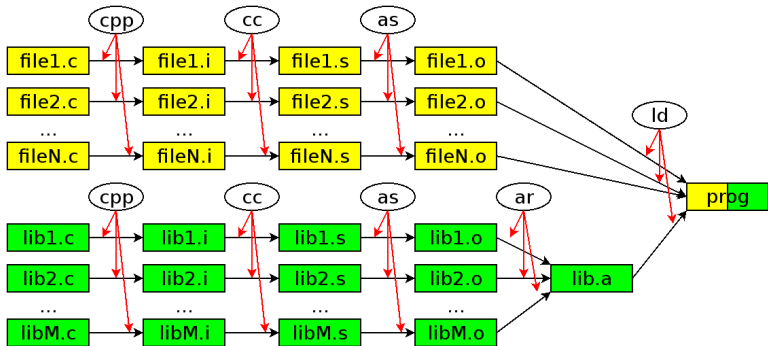
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

<http://sys.cs.fau.de/lehre/ss24>



Build-Chain



Schritte:

cpp: C-Pre-Prozessor Auflösung „#“-Anweisungen

cc: C-Compiler C-Code vereinfachen, nach Assembler wandeln

as: Assembler Assembler-Code nach Binär-Code wandeln

ar: Archiver Binär-Codes zu Libraries zusammenkleben

ld: Linker Binär-Codes und Libraries zusammenkleben



file.h:

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
```

file.c:

```
#include "file.h"
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

#ifdef USE_MAIN
int main(void) {
    REPEAT(10)
        OUTPUT(TEXT);
    return 0;
}
#else
#error "Problem..."
#endif
```



1. Schritt (#includes auflösen):

file.h:

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
```

file.c:

```
#include "file.h"
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

#ifdef USE_MAIN
int main(void) {
    REPEAT(10)
    OUTPUT(TEXT);
    return 0;
}
#else
#error "Problem..."
#endif
```

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

#ifdef USE_MAIN
int main(void) {
    REPEAT(10)
    OUTPUT(TEXT);
    return 0;
}
#else
#error "Problem..."
#endif
```



2. Schritt (#ifs auswerten):

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

#ifdef USE_MAIN
int main(void) {
    REPEAT(10)
    OUTPUT(TEXT);
    return 0;
}
#else
#error "Problem..."
#endif
```

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

int main(void) {
    REPEAT(10)
    OUTPUT(TEXT);
    return 0;
}
```



3. Schritt (Text ersetzen):

```
#define USE_MAIN
#define TEXT "hallo"
#define OUTPUT(x) \
    printf("%s\n", x)
#define REPEAT(n) \
    for (int i = 0; i < n; i++);

int main(void) {
    REPEAT(10)
        OUTPUT(TEXT);
    return 0;
}
```

file.i:

```
int main(void) {
    for (int i = 0; i < 10; i++);
        printf("%s\n", "hallo");
    return 0;
}
```



Berechnungen

- Berechnungen immer mit dem „größten“ Typ (mindestens „int“)
- ggf. Konvertierung beim Abspeichern des Resultats

```
uint8_t x;  
uint8_t y;  
double d;  
  
x = 2 * y + 3 * d;
```

```
uint8_t x;  
uint8_t y;  
double d;  
  
/* first product */  
int reg0 = (int) y;  
int reg1 = 2 * reg0;  
  
/* second product */  
double reg2 = d;  
double reg3 = 3.0 * reg2;  
  
/* sum */  
double reg4 = (double) reg1;  
double reg5 = reg4 + reg3;  
  
/* result */  
x = (uint8_t) reg5;
```



Typen

```
int a;  
int *b;  
int **c;  
  
int d[10];  
int *e[10];  
  
int f(int a);  
int *g(int a);  
  
int (*h)(int a);  
int *(*i)(int a);
```

```
const int a = 1;  
  
const int *b = &a;  
int * const c = NULL;  
const int * const d = b;  
  
volatile int e;  
  
volatile int *f;  
int * volatile g;  
volatile int * volatile h;  
  
const volatile int * const i = ...;
```

```
volatile int *(* const func)(void (*)(int *[])) = ...;
```



sizeof

```
int main(void) {
    uint32_t a, b[10], *c;
    struct {
        uint32_t x;
        uint32_t y;
    } d, e[10], *f;
    printf("%d\n", (int) sizeof(a)); /* printf("%zd\n", sizeof(a)); */
    printf("%d\n", (int) sizeof(b));
    printf("%d\n", (int) sizeof(c));
    printf("%d\n", (int) sizeof(d));
    printf("%d\n", (int) sizeof(e));
    printf("%d\n", (int) sizeof(f));
    return 0;
}
```

```
~> arch
x86_64
~> ./test
4
40
8
8
80
8
```



sizeof / Arrays

```
static void funcA(uint32_t *array) {
    printf("funcA: %zd\n", sizeof(array));
}
static void funcB(uint32_t array[]) {
    printf("funcB: %zd\n", sizeof(array)); /* <-- Warnung! */
}
static void funcC(uint32_t array[10]) {
    printf("funcC: %zd\n", sizeof(array)); /* <-- Warnung! */
}
int main(void) {
    uint32_t array[10];
    funcA(array); funcB(array); funcC(array);
    printf("func: %zd\n", sizeof(array));
    return 0;
}
```

```
~> arch
x86_64
~> ./test
funcA: 8
funcB: 8
funcC: 8
func: 40
```



Java-Version:

```
class Test {
    int val;

    int square() {
        return this.val * this.val;
    }
}

...
Test obj = new Test;

obj.val = 13;
System.println(obj.square());

...
```

C-Version:

```
struct Test {
    int val;
}

int square(struct Test *this) {
    return this->val * this->val;
}

...
struct Test *obj
    = malloc(sizeof(struct Test));
if (obj == NULL) { ... }
obj->val = 13;
printf("%d\n", square(obj));
free(obj);

...
```



C vs. JAVA

Java-Version:

```
class Test {  
    ...  
}  
...  
Test obj = new Test;  
obj.val = 13;  
System.println(obj.square());
```

C-Version:

```
typedef struct {  
    ...  
} Test;  
...  
Test *obj = NEW(Test);  
obj->val = 13;  
printf("%d\n", square(obj));  
DELETE(obj);
```

mit

```
#define NEW(type) ((type *) new(sizeof(type)))  
#define DELETE(ptr) free(ptr)  
  
void *new(size_t size) {  
    void *ptr = malloc(size);  
    if (ptr == NULL) {  
        perror("malloc");  
        exit(EXIT_FAILURE);  
    }  
    return ptr;  
}
```



Event Handling

```
static volatile uint8_t event1;
static volatile uint8_t event2;
...
ISR(..._vect) {
    ...
    event1 = 1;
}

ISR(..._vect) {
    ...
    event2 = 1;
}
...
void main(void) {
    /* Initialize events. */
    ...
    sleep_enable();

    /* Enable interrupts. */
    sei();
    ...
}
```

```
while (1) {
    /* Wait for event. */
    cli();
    while (! event1 && ! event2) {
        sei();
        sleep_cpu();
        cli();
    }
    sei();

    /* Handle event. */
    if (event1) {
        event1 = 0;
        ...
    }
    if (event2) {
        event2 = 0;
        ...
    }
}
}
```



Original-Code:

```
/* v-- "volatile" fehlt! */
static uint16_t counter;
...
ISR(...) {
    counter++;
}

...

cli();

while (counter < 100) {
    sei();
    sleep_cpu();
    cli();
}

sei();
...
```

Vereinfachter Code:

```
/* v-- "volatile" fehlt! */
static uint16_t counter;
...
ISR(...) {
    uint16_t reg;
    reg = counter;
    reg++;
    counter = reg;
}
...
uint16_t reg;
cli();
goto test;
loop:
    sei();
    sleep_cpu();
    cli();
test:
    reg = counter;
    if (reg < 100) goto loop;
    sei();
    ...
```



Vereinfachter Code:

```
...
uint16_t reg;

cli();
goto test;
loop:
sei();
sleep_cpu();
cli();
test:
reg = counter; /* <-- ! */
if (reg < 100) goto loop;
sei();
...
```

optimierter Code:

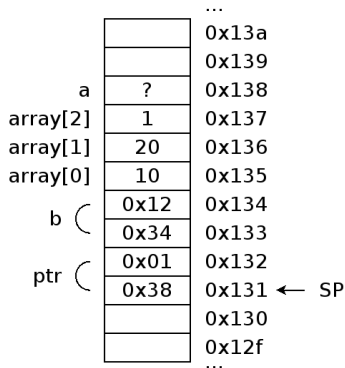
```
...
uint16_t reg;
reg = counter; /* <-- ! */
cli();
goto test;
loop:
sei();
sleep_cpu();
cli();
test:

if (reg < 100) goto loop;
sei();
...
```



Dynamische Speicherallokation – Stack

```
int main(void) {  
    uint8_t a;  
    uint8_t array[3] = { 10, 20, 1 };  
    uint16_t b = 0x1234;  
    uint8_t *ptr = &a;  
    ...  
}
```



- **Nur, wenn** ein System-Aufruf einen Fehler zurückmeldet, **dann** ist `errno` gesetzt => dann `perror/streerror` nutzen (**sonst nicht!**)
`errno` **nicht** gesetzt: `errno` **ist** gesetzt:

```
if (argc < 2) {  
    fprintf(stderr, "Usage: ...\n");  
    exit(EXIT_FAILURE);  
}
```

```
if (1024 <= strlen(name)) {  
    fprintf(stderr, "Bad name.\n");  
    exit(EXIT_FAILURE);  
}
```

```
FILE *fp = fopen(name, "r");  
if (fp == NULL) {  
    perror(name);  
    exit(EXIT_FAILURE);  
}
```

```
int *ptr = malloc(sizeof(int));  
if (ptr == NULL) {  
    perror(name);  
    exit(EXIT_FAILURE);  
}
```



```
static void err(const char *s) {
    fprintf(stderr, "%s\n", s);

    exit(EXIT_FAILURE);
}
```

errno **nicht** gesetzt:

```
if (argc < 2) {
    err("Usage: ...");
}
```

```
if (1024 <= strlen(name)) {
    err("Bad name.");
}
```

```
static void die(const char *s) {
    fprintf(stderr, "%s: %s\n",
            s, strerror(errno));
    exit(EXIT_FAILURE);
}
```

errno **ist** gesetzt:

```
FILE *fp = fopen(name, "r");
if (fp == NULL) {
    die(name);
}
```

```
int *ptr = malloc(sizeof(int));
if (ptr == NULL) {
    die(name);
}
```



Dateien

```
int main(void) {  
    printf("Hallo, Welt!\n");  
  
    return 0;  
}
```

oder

```
int main(void) {  
    FILE *fp = stdout;  
  
    fprintf(fp, "Hallo, Welt!\n");  
  
    return 0;  
}
```

```
~> ./test  
Hallo, Welt!
```

```
int main(void) {  
    /* Open file. */  
    FILE *fp = fopen("data.txt", "w");  
    if (fp == NULL) { ... }  
  
    /* Write to file. */  
    fprintf(fp, "Hallo, Welt!\n");  
  
    /* Close file. */  
    int ret = fclose(fp);  
    if (ret < 0) { ... }  
  
    return 0;  
}
```

```
~> ./test  
~> cat data.txt  
Hallo, Welt!
```



Verzeichnisse

```
/* Open directory. */
DIR *dp = opendir(".");
if (! dp) { ... }

while (1) {
    /* Read entry. */
    errno = 0;
    struct dirent *de = readdir(dp);
    if (! de && errno) { ... }
    if (! de) {
        break;
    }
    /* Process entry. */
    printf("%s\n", de->d_name);
}

/* Close directory. */
(void) closedir(dp);
```

```
~> ./test
.
..
test
test.c
~>
```



**Vielen Dank für's Mitmachen!
Vielen Dank für die gute Atmosphäre!**

Viel Erfolg!

