

Handling von Variablen

- Prozess- / Speichermodell:
 - Globale Variablen
 - Heap
 - Stack
- Stackmodell
- Parameterübergabe
- Rückgabewert auf Stack
- Rücksprungadresse auch auf Stack!
- "Runtime" Parameterhandling von C Implementiert!

cpp, der Prä-Prozessor

- Definition / Ersetzung

```
#define kuerzel irgend_ein_langer_text
```

- Test

```
#ifdef kuerzel
```

```
...
```

```
#endif
```

- Inklusion

```
#include <namen_einer_anderen_datei.txt>
```

Einfache Deklarationen

- Einfache Typen:

```
int, char, long, ...
```

- Deklaration:

```
int zahl;  
char buchstabe;
```

- Funktionsdefinition:

```
int f(int z) { return z + z; }
```

- Parameterübergabe *immer* “by value” auf dem Stack:

```
f(zahl);
```

Pointer

- Pointer:

```
int*, char*, long*, ...
```

- Deklaration

```
int*   zahl_p;  
char*  buchstaben_p;    /* oft ein Pointer auf  
                        einen String!    */
```

- Funktionsdeklaration:

```
void f(int* z_p);
```

- Übergeben wird *immer* Wert (d.h. hier wird **Adresse** als *Wert* übergeben):

```
f(zahl_p);
```

Dereferenzieren

- mehrere Bedeutungen des Stern Operators
- Deklaration eines Pointers:

```
char* buchstaben_p = "p";
```

(Achtung: String!)

- Dereferenzierung:

```
char buchstabe;  
buchstabe = *buchstaben_p;
```

- Operatoren Reihenfolge, Achtung "C Code":

```
for (; *to++ = *from++; ) ;
```

"++" bindet stärker als "*"

Referenzieren

- Operator "&":

```
int z;  
int* z_p;  
  
z=7;  
z_p = &z;  
*z_p = 9;
```

- `void f(int* i);`

```
int z;  
f(&z);
```

- Achtung:

```
int z1, z2, z3;  
z3 = z1 & z2;    /* bitweise und-Verknüpfung */
```

Arrays

- `char c[20];`

oft als String verwendet. Dann mit `'\0'` abgeschlossen:

```
char ca[2] = "a";  
char cb[2];
```

```
cb[0] = 'b';  
cb[1] = '\0';
```

- beachte verschiedene Anführungszeichen.

Arrays

- Jedoch:

```
$ cat -n kak.c
    1 void f(char a[]);
    2 void g(char a[20]);
    3
    4 main() {
    5     char a[];
    6 }

$ gcc kak.c
kak.c: In function `main':
kak.c:5: array size missing in `a'
```


Strukturen

```
struct my_struct {
    int a;
    char b;
} s1;

struct my_struct s2;

struct my_struct* s2_p;

main() {
    s1.a = 2;
    s2.b = 'z';

    s2_p = &s2;
    (*s2_p).a = 3;
    s2_p->a = 3;    /* gleich wie eben, aber
                    leichter zu lesen */
}
```

Typedef

- Alias, "neuen `einfachen' Typ definieren"

-> Typenprüfung

-> Abstraktion

```
typedef struct my_struct* my_struct_p;
```

```
my_struct_p m_p;
```

Funktionspointer

```
void f(int a) {  
    printf("the number is %d\n", a);  
}  
  
void exec(void function(int a), int argument) {  
    function(argument);  
}  
  
main() {  
    exec(f, 3);  
}
```

Kombinationen

- möglich!
- Pointer auf Pointer
- Pointer auf Struktur
- Arrays von Pointern auf Strukturen etc.

Funktionsaufrufe

```
int b;  
int* b_p;
```

```
b_p = &b;
```

Wert übergeben:

```
void f(int x);  
f(b);
```

Funktionsaufrufe

Pointer auf Wert übergeben

- Wert ändern
- Kopieren auf Stack sparen

```
void f_(int* x) { *x = 7; }
f_(b_p);
```

```
struct s {
    ...
} s_i;
```

```
void g(struct s ss);    /* hier wird die ganze */
g(s_i);                /* Struktur als Wert */
                        /* übergeben */
```

```
void g_(struct s* sp); /* hier wird jedoch */
g(&s_i);                /* nur ein Pointer */
                        /* übergeben */
```

Funktionsaufrufe

Pointer auf Pointer übergeben

- Pointer umlenken
- Funktion will allozierten Bereich zurückgeben
- Achtung Deallokation!

```
int* b_p;

void f(int** b_pp) {
    int* i_p;

    i_p = (int* )malloc(sizeof(int));
    *i_p = 7;

    *b_pp = i_p;
}

f(&b_p);
```

Werkzeuge: gcc

gcc

C Compiler

ruft automatisch Linker auf

```
$ gcc prog.c
$ ls
gcc.c      a.out
$ ./a.out
[compiliertes prog.c wird ausgeführt]
```

mehr Info:

```
$ man gcc; info gcc
```


Werkzeuge: make

make

automatisches Erstellen von Programmen (und anderem)
verwendet die Datei Makefile

```
$ cat Makefile
prog: prog.c
    gcc prog.c -o prog
$ make
[ Compilation beginnt, "prog" wird erstellt ]
```

Werkzeuge: make

```
prog: prog.c
    gcc prog.c -o prog
```

- **prog** ist ein Ziel bzw. eine Datei
- die Erstellung von **prog** hängt vom Vorhandensein von **prog.c** ab
- das heisst auch, dass **prog** neu erstellt werden muss, sobald sich **prog.c** ändert
- das Rezept, um **prog** zu erstellen folgt in der nächsten Zeile und ist eine Shell Anweisung
- die Rezept Zeile **muss** zwingend mit einem Tabulator anfangen
- im obigen Rezept sieht man, dass der Compiler das Programm namens **prog** erstellt - somit ist das Ziel erreicht.