

Kurseinheit 3: Funktionen

1. Einsatzbereiche für Funktionen
2. Deklaration, Definition und Aufruf
3. Gültigkeitsbereich von Variablen
4. Bibliotheken
5. Unsichere Funktionen
6. Sicheres Programmieren

Übersicht KE 3

Lehrveranstaltung Ingenieur-Informatik – 2 SWS/2 Credits: EI1, EI+1, MKA1, MK+1, EI3nat3
Lehrveranstaltung Programmierung 2 (Teil C) – 2 SWS/2 Credits: AI2

Unterrichtsdauer für diese Kurseinheit: 90 Minuten

Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: Kapitel 4

Zusatzthemen: Bibliotheken, insbesondere statische Bibliotheken, unsichere Funktionen
sowie Sicheres Programmieren

1. Einsatzbereiche für Funktionen

3

DRY-Prinzip

DRY: Don't Repeat Yourself

Bei der Programmierung werden oft immer wieder die gleichen Programmteile benötigt. Falls diese dann auch mehrfach im Programm vorkommen, ergeben sich folgende Probleme:

- Durch Kopieren der gleichen Programmteile werden Fehler mitkopiert.
- Die eigentlich gleichen Programmteile werden nicht überall bei Anpassungen geändert.
- Das Programm wird dadurch größer und unübersichtlicher.

Dies ist zu vermeiden (DRY).

Programmteile, die mehrfach benötigt werden, sind in einer Funktion zu implementieren. Diese Funktion kann dann überall im Programm aufgerufen werden. Fehlerbehebungen (Bug-Fixes) und Erweiterungen sind somit nur an einer Stelle notwendig.

Diese nützlichen Funktionen können dann einfach auch in andere Projekte integriert werden. Projektübergreifende Funktionen sind oft in Bibliotheken enthalten.

Schlagwort hierzu: **Wiederverwendbarkeit**

1. Einsatzbereiche für Funktionen

4

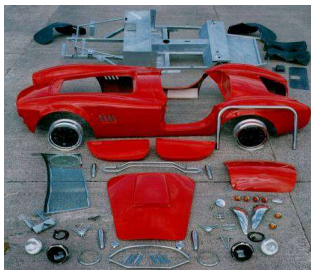
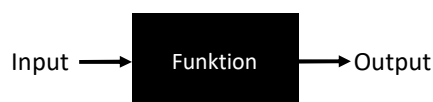
Wiederverwendbarkeit

Aus einzelnen Teilen (Funktionen) kann ein gesamtes Fahrzeug (Programm) gebaut werden.

Sind diese Teile gut getestet und qualitativ hochwertig und passen diese auch zueinander (Schnittstelle), so gestaltet sich der Bau eines Fahrzeugs recht einfach.

Oft muss **nur** die Schnittstelle verstanden werden (Was ist wo anzuschließen?).

Man spricht dabei von einem Black-Box-Verhalten:



2. Deklaration, Definition und Aufruf

5

Ohne Parameterübergabe und Ergebnissrückgabe

```
#include <stdio.h>

// Function Declaration
void PrintName(void);

int main(void)
{
    // Function Call
    PrintName();

    return 0;
}

//Function Definition
//"Implementation"
void PrintName(void)
{
    printf("Cooper\n");
}
```

Deklaration: Eine Funktion muss erst **deklariert** werden. void bedeutet „Leere“ oder „Nichts“. Hier werden keine Parameter übergeben und es wird auch kein Ergebnis zurückgeliefert (Ergebnissrückgabe).

Es sollte ein selbstsprechender Funktionsname (hier PrintName) verwendet werden. Siehe hierzu C-Coding Styleguide F11 II.

Aufruf: Die neue Funktion kann nun von überall aufgerufen werden. Nachdem die neue Funktion abgearbeitet ist, wird wieder zurück (Zeile nach Aufruf) gesprungen.

Die **Definition** (auch oft Implementierung genannt) der Funktion sollte unterhalb von main stattfinden.

2. Deklaration, Definition und Aufruf

6

Ohne Parameterübergabe und Ergebnissrückgabe - Modellierung

```
#include <stdio.h>

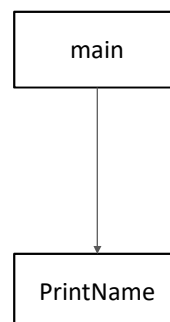
// Function Declaration
void PrintName(void);

int main(void)
{
    // Function Call
    PrintName();

    return 0;
}

//Function Definition
//"Implementation"
void PrintName(void)
{
    printf("Cooper\n");
}
```

Die Architektur von Programmen muss auch oft dokumentiert werden. Für C eignet sich das **Structure Chart** (dt. Strukturdiagramm). Funktionen sind Blöcke, Aufrufe sind Pfeile. Der Rücksprung ergibt sich implizit und wird nicht eingezeichnet.



2. Deklaration, Definition und Aufruf

7

Mit Parameterübergabe und ErgebnISRückgabe

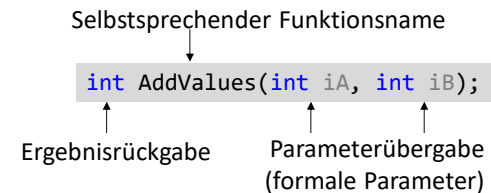
```
// Function Declaration
int AddValues(int iA, int iB);

int main(void)
{
    int iRes;
    int iVal1 = 3;
    // Function Call
    iRes = AddValues(5, 13);
    iRes = AddValues(iVal1, 73);

    return 0;
}

//Function Definition
int AddValues(int iA, int iB)
{
    int iResult = iA + iB;
    return iResult;
}
```

Funktionen können auch einen Rückgabewert (ErgebnISRückgabe) und mehrere Übergabewerte (Parameterübergabe oder formale Parameter) haben. **return** liefert Rückgabewert.



Bei der Deklaration sind die Variablennamen optional.

Hinweis: Die Schnittstelle muss unbedingt erfüllt werden. Anzahl und Datentyp der Übergabeparameter müssen immer gleich sein. ErgebnISRückgabe ggf. casten.

2. Deklaration, Definition und Aufruf

8

Mit Parameterübergabe und ErgebnISRückgabe - Modellierung

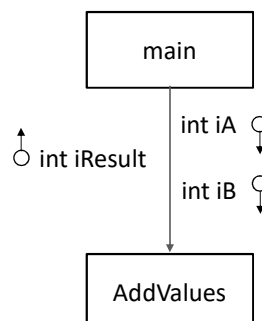
```
// Function Declaration
int AddValues(int iA, int iB);

int main(void)
{
    int iRes;
    int iVal1 = 3;
    // Function Call
    iRes = AddValues(5, 13);
    iRes = AddValues(iVal1, 73);

    return 0;
}

//Function Definition
int AddValues(int iA, int iB)
{
    int iResult = iA + iB;
    return iResult;
}
```

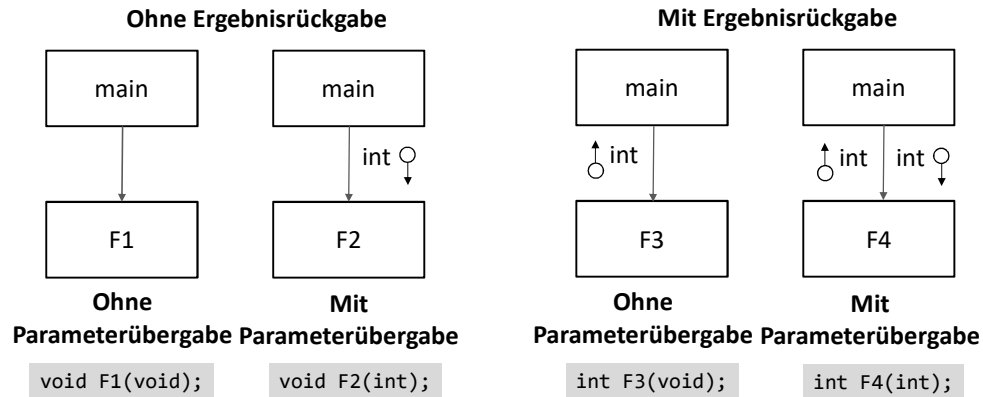
Beim **Structure Chart** ist die Parameterübergabe und die ErgebnISRückgabe als Datentyp anzugeben. *Optional* können zur besseren Lesbarkeit auch noch die Variablennamen verwendet werden.



2. Deklaration, Definition und Aufruf

9

Alle Kombinationen im Überblick



Bei der Parameterübergabe können auch mehr als eine Variable übergeben werden (hier wurde nur eine Variable verwendet.). Auch können alle Datentypen für die Ergebnissrück- und Parameterübergabe verwendet werden (hier wurde nur int verwendet).

2. Deklaration, Definition und Aufruf

10

return

Das Schlüsselwort return gibt einen Wert an die aufrufende Funktion zurück. Hierdurch geschieht auch der Rücksprung.

```
int AddValues(int iA, int iB)
{
    int iResult = iA + iB;
    return iResult;
}
```

Kurzform
→
ohne
Hilfsvariable

```
int AddValues(int iA, int iB)
{
    return (iA + iB);
}
```

In C sind mehrere (bedingte) returns in einer Funktion möglich. Der hier verwendete C-Coding Styleguide verbietet dies aber. return ist immer die letzte Anweisung bei einer Funktion mit Ergebnissrückgabe.

Funktionen ohne Ergebnissrückgabe können am Ende ein return verwenden, dies ist aber nicht zwingend notwendig.

```
void PrintName(void)
{
    printf("Cooper\n");
}
```

äquivalent
↔

```
void PrintName(void)
{
    printf("Cooper\n");
    return;
}
```

3. Gültigkeitsbereich von Variablen

11

Global und lokal

In Abhängigkeit wo die Variablen definiert wurden, haben diese einen unterschiedlichen Gültigkeitsbereich, in welchem sie überhaupt verwendet werden können. Oft wird hier auch der Begriff der „Sichtbarkeit“ verwendet.

Grundsätzlich lassen sich erst einmal lokale von globalen Variablen unterscheiden. Lokale Variablen können nur in dem Block {}, in dem sie definiert sind, verwendet werden. Globale Variablen können im gesamten Programm verwendet werden – in dieser Lehrveranstaltung soll auf globale Variablen zukünftig verzichtet werden.

```
int iBias = 5; //global

int main(void)
{
    auto int iRes; //local
    iRes = AddValues(5 + iBias);
    return 0;
}
```

Das Keyword auto wird meist weggelassen. Es ist somit optional.

Auf die globale Variable iBias kann von allen Funktionen aus (nicht nur von main) zugegriffen werden.

Auf die lokale Variable iRes kann hier nur von main aus zugegriffen werden, da diese darin definiert wurde.

3. Gültigkeitsbereich von Variablen

12

Speicherklassen

Speicherklassen geben Auskunft darüber, wann Speicherplatz für Variablen reserviert wird und wie lange der Speicherplatz gültig ist. In C gibt es vier Schlüsselworte für Speicherklassen.

Schlüsselwort	Verwendung	Speicherplatz	Initialisierung
auto	Lokale Variable	Stack	zufällig
static	Lokale Variable die ihren Wert behält oder modulglobale Variable	Globale Daten	0
extern	Zugriff auf eine globale Variable von einem anderen Modul aus.	Globale Daten	0
register	Wird in einem Prozessorregister gehalten.	Register	zufällig

In dieser KE wird jetzt nur auto und static (lokal) behandelt.

3. Gültigkeitsbereich von Variablen

13

Vergleich static versus auto

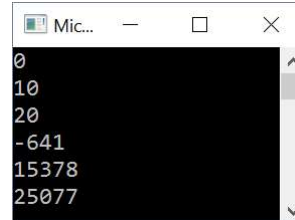
```
TestStatic();
TestStatic();
TestStatic();

TestLocal(); DoSomethingOnStack();
TestLocal(); DoSomethingOnStack();
TestLocal();
```

```
void TestStatic(void)
{
    static int iLocalStatic;

    printf("%d\n", iLocalStatic);
    iLocalStatic += 10;
}
```

Statische Variablen sind mit 0 vorinitialisiert und behalten ihren einmal zugewiesenen Wert.



```
0
10
20
-641
15378
25077
```

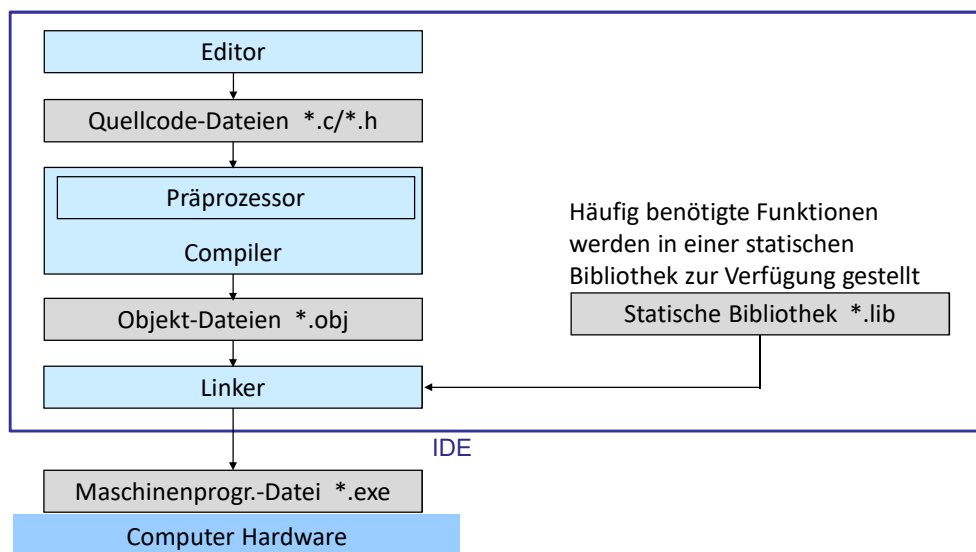
```
void TestLocal(void)
{
    auto int iLocalAuto;
    //equivalent to int iLocalAuto
    printf("%d\n", iLocalAuto);
    iLocalAuto = 10;
}
```

Auto Variablen sind mit einem zufälligen Wert initialisiert. Keyword auto kann weggelassen werden.

4. Bibliotheken

14

Prinzip



4. Bibliotheken

15

Drei Arten von statischen Bibliotheken

Bibliotheken, die der C-Standard vorgibt:

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

Es wird automatisch die hierzu notwendige statische Bibliothek eingebunden. Funktionen z.B. printf() und scanf()

Code kann mit jedem Compiler übersetzt werden -> Compiler/IDE unabhängig

Bibliotheken, die der Compilerhersteller mitliefert:

```
#include <conio.h>
```

Es wird automatisch die hierzu notwendige statische Bibliothek eingebunden. Funktionen z.B. _getch() (getchar ohne Echo)

Code kann **nicht** mit jedem Compiler übersetzt werden -> Compiler/IDE abhängig

Bibliotheken, die selbst erstellt wurden:

```
#include "Utilities.h"
```

Es muss manuell die hierzu notwendige statische Bibliothek eingebunden werden. Funktionen z.B. _gotoxy() und _clrscr();

4. Bibliotheken

16

Einbinden einer eigenen statischen Bibliothek

Es werden zwei Dateien benötigt:

- Headerdatei, die allerdings mit "" in der Quellcode-Datei *.c inkludiert wird. Headerdatei ist vorläufig in das Verzeichnis des Sourcecodes zu kopieren.
- Statische Bibliothek, die beim Linken hinzugefügt wird. Statische Bibliothek ist vorläufig in das Verzeichnis des Sourcecodes zu kopieren.

Properties des Projektes

->Configuration Properties->

->Linker -> Input

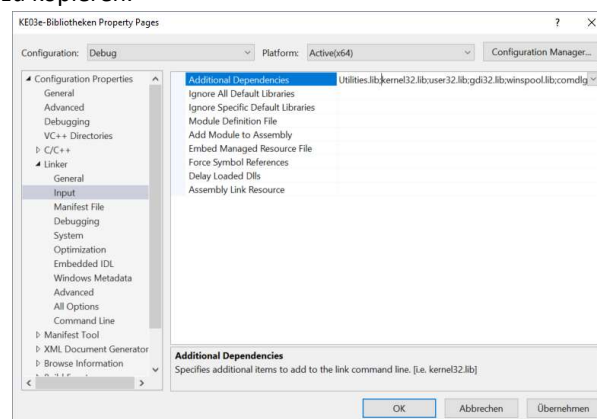
Bei Additional Dependencies

Utilities.lib;

einfügen und OK drücken

Wichtig: Die Einstellung gilt nur für die gewählte Konfiguration (hier Debug).

All Configurations für alle alle Konfigurationen anwählen.



4. Bibliotheken

17

Beispiel

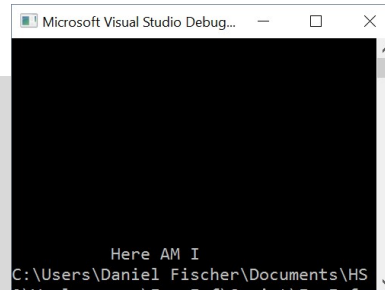
```
#include <stdio.h>    // C-Standard lib
#include <conio.h>     // Microsoft VS lib
#include "Utilities.h" // Own Utilities lib

int main(void)
{
    int iKey;

    printf("Please press ANY key:"); // C-Standard lib
    iKey = _getch();                // Microsoft VS lib

    _clrscr();                      // Own Utilities lib
    _gotoxy((short int)10, (short int)10); // Own Utilities lib
    printf("Here AM I");           // C-Standard lib

    return 0;
}
```



5. Unsichere Funktionen

18

Beispiele für unsichere Funktionen

Einige Funktionen aus Bibliotheken gelten als unsicher (Details siehe später). Beispiele:

- scanf()
- gets() – wurde sogar aus dem C11 Standard entfernt.

MSVS weist bei Benutzung dieser Funktionen aus, dass diese unsicher sind und verweist auf sichere Varianten. Diese gehören nicht zum Standard und damit wäre der Quellcode nicht mehr auf einem anderen Compiler compilierbar. Zudem wird eine Warnung ausgegeben.

In diesem Kurs soll nun auf gets() verzichtet werden und es soll nur noch die sichere Funktion scanf_s() verwendet werden. scanf_s() ist allerdings nur unter MSVS verfügbar.

Beispiel:

```
scanf("%d", &iVal); // Warning/Error
scanf_s("%d", &iVal); // No Warning/no Error
```

5. Unsichere Funktionen

19

Anwendung von scanf_s

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

int main(void)
{
    int iVal;
    float fVal;
    double dVal;
    char cVal;
    char acString[10];

    scanf("%d", &iVal);
    scanf_s("%d", &iVal);
    scanf_s("%f", &fVal);
    scanf_s("%lf", &dVal);
    scanf_s(" %c", &cVal, 1); // SPACE TRICK
    scanf_s("%9s", &acString, (unsigned int)_countof(acString));

    return 0;
}
```

Bei scanf_s muss berücksichtigt werden, dass bei den Formatbezeichnern %c und %s ein weiterer Parameter übergeben werden muss.

5. Sicheres Programmieren

20

Maßnahmen Einstellungen Compiler

Es gibt verschiedene Maßnahmen, die Fehler/schlechten Quellcode finden oder sogar verhindern. Neben speziellen Softwarewerkzeugen kann aber auch der **Compiler** so eingestellt werden, dass mögliche Fehler oder schlechter Quellcode gefunden werden. Oft gibt dieser dann **Warnungen** oder sogar **Fehler** aus.

Für **All Configurations** einstellen.

Statische Analyse: Überprüfung während dem Compilieren

Properties des Projektes, -> Configuration Properties -> C/C++ -> General

- **Warning Level** auf **Level4 (/W4)** stellen
- **SDL checks** auf **Yes (/sdl)** stellen. (SDL: Microsoft Security Development Lifecycle)

Dynamische Analyse: Überprüfung bei der Programmausführung

Properties des Projektes, -> Configuration Properties -> C/C++ -> Code Generation

- **Basic Runtime Checks** auf **Default** stellen. In Debug führt dies zu **(/RTC1, equiv. To /RTCsu)/RTC1** und in Release ist dies dann deaktiviert.
- **Security Check** auf **Enable Security Check (/GS)** stellen

5. Sicheres Programmieren

21

Weitere Maßnahmen

Handhabung Warnungen wie Fehler:

Bei Programmfehlern (error) wird kein ausführbares Programm generiert. Erzeugt der Compiler beim Übersetzen nur Warnungen (warnings) so wird ein möglicherweise fehlerhaftes ausführbares Programm generiert. Der Programmierer „übersieht“ oft die Warnungen, die im *Output* oder *Error List* Fenster ausgegeben werden.

Abhilfe:

Properties des Projektes, -> Configuration Properties -> C/C++ -> General

- **Treat Warning As Errors** auf **Yes (/WX)** stellen

C-Coding Styleguide: „Programmierrichtlinien“

Diese legen fest, wie programmiert werden muss. Nur so ist sichergestellt, dass der Quellcode einheitlich und fehlerfreier ist. Der Quellcode ist auch besser wartbar. Jede professionelle Firma gibt einen solchen C-Coding Styleguide vor. Darin sind dann unter anderen auch Regeln wie „Handhabung Warnungen wie Fehler“ vorgegeben. Der C-Coding Styleguide ist im Moodlekurs hinterlegt und ist nun **ab KE 4**, in den **Labortests** und in der **Prüfung anzuwenden**. Bei Nichteinhaltung erfolgen in der Prüfung und in den Labortests **Punktabzüge**.

Zusammenfassung KE 3

22

Behandelte Schlüsselwörter in KE 3

Schlüsselwörter C89:

auto✓	do	goto	signed✓	unsigned✓
break	double✓	if	sizeof✓	void✓
case	else	int✓	static✓	volatile✓
char✓	enum	long✓	struct	while
const✓	extern✓	register✓	switch	
continue	float✓	return✓	typedef	
default	for	short✓	union	

Schlüsselwörter ab C99:

_Bool✓	_Complex✓	_Imaginary✓	inline	restrict
--------	-----------	-------------	--------	----------

Schlüsselwörter ab C11:

_Alignas	_Alignof	_Atomic	_Generic	_Noreturn
_Static_assert	_Thread_local			