

## Kurseinheit 9: File I/O und Präprozessor

1. File I/O
2. csv-Dateien
3. Präprozessor
4. Funktionsmakros und Inlinefunktionen

## Übersicht KE 9

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 9 und 10

Zusatzthemen: csv-Dateien und inline

## 1. File I/O

3

### Persistenz

Unter Persistenz versteht man das dauerhafte Speichern von Daten. Im Gegensatz zu persistenten Daten stehen transiente Daten („vorübergehende“ Daten) nach dem Programmneustart nicht mehr zur Verfügung. Bisher wurde in diesem Kurs nur mit transienten Daten gearbeitet.

Grundsätzlich gibt es zwei Möglichkeiten zur Persistenz von Daten:

- Speichern und Lesen von **Dateien (Files)**
- Verwendung eines **Datenbankmanagementsystems DBMS** (Oracle, MS SQL Server, MySQL). Intern speichert dieses DBMS die Daten auch in Dateien, der interne Aufbau dieser Dateien ist für den Programmierer nicht wichtig. Dieser benutzt nur API-Funktionen (SQL-Befehle) um auf die Daten zuzugreifen.

Dies alles funktioniert natürlich nur, wenn der Rechner auch ein Dateisystem besitzt. Bei Embedded Systemen ist dies aber nicht der Fall. Dann müssen Daten in einem EEPROM abgelegt werden (siehe Embedded Systems 2).

**Inhalt in dieser KE ist u.a. das Speichern und Lesen von Dateien (File I/O).**

## 1. File I/O

4

### Prinzip

Es ist <stdio.h> zu inkludieren. Darin sind die File I/O-Funktionen und ebenso der Datentyp FILE deklariert. File I/O-Funktionen beginnen mit einem „f“ für file. Hier im Beispiel sind dies fopen\_s, fputs und fclose. Es soll hier die **sichere Funktion fopen\_s** verwendet werden.

```
FILE* pFMessage = NULL; // pointer to file
errno_t iErrorCode;
```

```
typedef struct _iobuf
{
    void* _Placeholder;
}FILE;
```

```
iErrorCode = fopen_s(&pFMessage, "c:\\temp\\students.txt", "w");
```

Warum zwei \ ?

```
if (iErrorCode == 0)
{
    fputs("The students are motivated", pFMessage);
    fclose(pFMessage); // Close the file
    pFMessage = NULL; // For sanity checks
}
else
{
    printf("Can not open file: ErrorCode: %i\n", iErrorCode);
}
```

## 1. File I/O

5

### Datei öffnen mit `fopen_s` (1)

```
errno_t fopen_s (FILE** ppFile, const char* pccFilename,  
                const char* pccModus);
```

**Rückgabewert:** Falls `fopen_s` ohne Fehler funktioniert hat, wird eine 0 zurückgegeben. Falls ein Fehler auftrat, steht dann die Fehlernummer im Rückgabewert. Details unter:  
<https://docs.microsoft.com/de-de/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=vs-2019>

**ppFile** ist ein Zeiger auf den Filepointer. Wenn `fopen_s` fehlschlägt, ist dieser NULL.

Der Dateiname (**pccFilename**) kann dabei wie folgt gewählt werden:

- Absoluter Pfad mit Dateiname (z.B. `c:\temp\test.txt`)
- Relativer Pfad mit Dateiname ausgehend vom aktuellen Arbeitsverzeichnis (z.B. `..\files\test.txt`)
- Nur Dateiname, dann wird allerdings nur im aktuellen Arbeitsverzeichnis gesucht.

Letztere Variante scheint die einfachste Variante zu sein. Es ist allerdings oft unklar, was das aktuelle Arbeitsverzeichnis ist. Wird mit der MSVS IDE gearbeitet, ist dies das Projektverzeichnis.

## 1. File I/O

6

### Datei öffnen mit `fopen_s` (2)

```
errno_t fopen_s (FILE** ppFile, const char* pccFilename,  
                const char* pccModus);
```

pccModus	Access
"r"	Opens for reading. If the file does not exist or cannot be found, the <b>fopen_s</b> call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending) without removing the end-of-file (EOF) marker before new data is written to the file. Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. The appending operation includes the removal of the EOF marker before new data is written to the file. The EOF marker is not restored after writing is completed. Creates the file if it does not exist.

<https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/fopen-s-wfopen-s?view=vs-2019>

## 1. File I/O

7

### Datei öffnen mit fopen\_s (3)

```
errno_t fopen_s (FILE** ppFile, const char* pccFilename,  
                const char* pccModus);
```

Bewirkt	r	w	a	r+	w+	a+
Datei ist lesbar.	x			x	x	x
Datei ist beschreibbar.		x	x	x	x	x
Datei ist nur am Dateiende beschreibbar.			x			x
Existierender Dateiinhalt geht verloren.		x			x	

[http://openbook.rheinwerk-](http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/016_c_ein_ausgabe_funktionen_005.htm#mj2f57f419fdeadcd1c7dd4e001616d21a)

[Verlag.de/c\\_von\\_a\\_bis\\_z/016\\_c\\_ein\\_ausgabe\\_funktionen\\_005.htm#mj2f57f419fdeadcd1c7dd4e001616d21a](http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/016_c_ein_ausgabe_funktionen_005.htm#mj2f57f419fdeadcd1c7dd4e001616d21a)

In pccModus kann auch noch ein Translationsmode eingefügt werden:

Mit einem „t“ wird die Datei im Textmodus geöffnet, mit einem „b“ in einem Binärmodus.

Wird keiner dieser Buchstaben in pccModus verwendet, so wird die Defaulteinstellung verwendet (Textmodus).

## 1. File I/O

8

### Datei schließen mit fclose

```
int fclose(FILE* pFile);
```

**Rückgabewert:** 0 falls alles in korrekt war, EOF falls Fehler auftrat.

**Übergabeparameter pFile:** Gültiger Filepointer (FILE\*)

Betriebssysteme begrenzen intern die Anzahl an Files, die ein Prozess (Programm in Ausführung) maximal geöffnet haben kann. Die Laufzeitbibliothek von C (stdio.h) begrenzt dies auf 512 Files pro Prozess.

Daher sollten geöffnete Dateien wieder mit fclose geschlossen werden, sobald diese nicht mehr benötigt werden.

```
if (iErrorCode == 0)  
{  
    fclose(pFMessage); // Close the file  
    pFMessage = NULL; // For sanity check  
}  
else  
{  
}
```

fclose ist in dem Programmzweig aufzurufen, in dem die Datei geöffnet werden konnte.

Der Aufruf mit einem NULL-Zeiger führt zu einer Exception. Nach close ist der Zeiger auch auf NULL zu setzen (Sanity Check).

## 1. File I/O

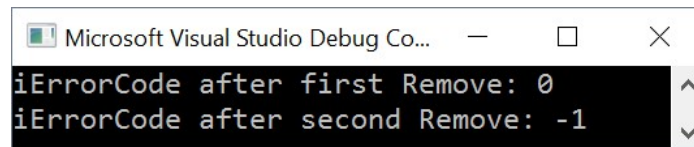
9

### Dateien löschen mit remove

```
int remove(const char* pccFileName);
```

Oft muss ein Programm auch Dateien löschen können. Dies geschieht mit der Funktion `remove`. War der Aufruf erfolgreich, wird eine 0 zurückgegeben, ansonsten ungleich 0.

```
iErrorCode = remove("c:\\temp\\students.txt");  
printf("iErrorCode after first Remove: %d\\n", iErrorCode);  
iErrorCode = remove("c:\\temp\\students.txt");  
printf("iErrorCode after second Remove: %d\\n", iErrorCode);
```



## 1. File I/O

10

### In Datei schreiben

Es gibt vielfältige Funktionen, um in eine Datei zu schreiben. Diese sind bereits als Konsolenausgabe bekannt. Es ist nur ein `f` (für File) vorangestellt und in den Übergabeparametern ist noch der Filepointer `pFile` mitzugeben.

```
int fputc(int iCharacter, FILE* pFile);
```

Fehlerfall: EOF wird zurückgeben, sonst ASCII Code des Zeichens.

```
int fputs(const char* pccString, FILE* pFile);
```

Fehlerfall: EOF wird zurückgeben, sonst nicht negativer Wert.

```
int fprintf (FILE* pFile, const char * pccFormat, ... );
```

Fehlerfall: Es wird eine negative Zahl zurückgegeben, sonst die Anzahl geschriebener Zeichen.

In der Struktur `FILE` befindet sich eine Art Schreib-/Lesezeiger, der auf eine Position in der Datei zeigt. Mit jedem Schreib- oder Lesezugriff wird dieser Zeiger intern verschoben. Dieser ist neben weiteren Daten in der `FILE`-Struktur bzw. hinter dem Element `void* _Placeholder` hinterlegt.

## In Datei schreiben - Beispiel

```

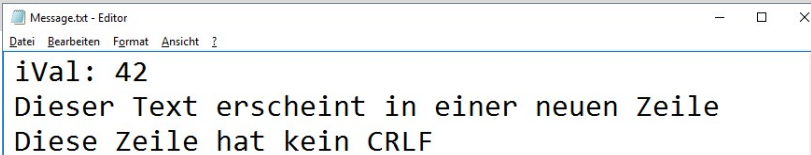
iErrorCode = fopen_s(&pFMessage, "Message.txt", "w");

if (iErrorCode == 0)
{
    iVal = 42;
    fprintf(pFMessage, "iVal: %i\n", iVal);
    fprintf(pFMessage, "Dieser Text erscheint in einer neuen Zeile\n");
    fprintf(pFMessage, "Diese Zeile hat kein CRLF");
    fclose(pFMessage); // Close the file
    pFMessage = NULL;
}
else
{
    printf("Can not open file: ErrorCode: %i\n", iErrorCode);
}

```

Ohne Pfadangabe: Datei wird im aktuellen Arbeitsverzeichnis geöffnet (MSVS: Projektverzeichnis)

Geöffnet mit notepad.



## Aus Datei lesen

Auch beim Lesen aus Dateien gibt es unterschiedliche Funktionen. Im Labor soll hauptsächlich fgets zur Anwendung kommen.

```
int fgetc(int iCharacter, FILE* pFile);
```

Fehlerfall: EOF wird zurückgeben, sonst ASCII Code des eingelesenen Zeichens.

```
char* fgets(char* pcStr, int iMaxNumber, FILE* pFile);
```

Eine komplette Zeichenkette wird eingelesen, bis das CRLF erkannt wird oder iMaxNumber-1 Zeichen eingelesen wurden. iMaxNumber-1 deshalb, weil noch ein EOS eingefügt werden muss. Falls eine komplette Zeile eingelesen wurde, so ist das **CRLF Bestandteil der Zeichenkette** auf die pcStr zeigt. Rückgabewert ist der Zeiger auf die eingelesene Zeichenkette oder im Fehlerfall eine NULL.

```
char* fscanf_s(FILE* pFile, const char * pccFormat, ... );
```

<https://docs.microsoft.com/de-de/cpp/c-runtime-library/reference/fscanf-s-fscanf-s-l-fwscanf-s-fwscanf-s-l?view=vs-2019>

Mit jedem Lesezugriff wird der interne Schreib-/Lesezeiger verschoben.

## Erkennung Dateiende (1)

Möglichkeit 1: Rückgabe von fgets überprüfen

```
iErrorCode = fopen_s(&pFMessage, "Message.txt", "r");
if (iErrorCode == 0)
{
    do
    {
        pcReadStrReturn = fgets(acLine, 255, pFMessage);
        if (pcReadStrReturn != NULL)
        {
            printf("%s", pcReadStrReturn);
        }
    } while (pcReadStrReturn != NULL);
    fclose(pFMessage); // Close the file
    pFMessage = NULL;
}
else
{
    printf("Can not open file: ErrorCode: %i\n", iErrorCode);
}
```

Ist pcReadStrReturn NULL, so steht in acLine noch der vorherige Wert!

## Erkennung Dateiende (2)

Möglichkeit 2: Funktion int feof(FILE pfFile\*) verwenden

feof gibt eine 0 zurück, falls das Dateiende noch nicht erreicht ist.

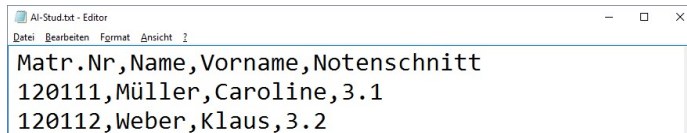
```
iErrorCode = fopen_s(&pFMessage, "Message.txt", "r");
if (iErrorCode == 0)
{
    iEOF = feof(pFMessage);
    while (iEOF == 0)
    {
        fgets(acLine, 255, pFMessage);
        printf("%s", acLine);
        iEOF = feof(pFMessage);
    }
    fclose(pFMessage); // Close the file
    pFMessage = NULL;
}
else
{
    printf("Can not open file: ErrorCode: %i\n", iErrorCode);
}
```

## 2. csv-Dateien

15

### Formatbeschreibung

csv-Dateien (Comma Separated Values) sind in der Anwendungsentwicklung sehr wichtig. Sie dienen zum Datenaustausch zwischen verschiedenen Anwendungen (Import/Export) und können sogar von Tabellenkalkulationsprogrammen wie Excel gelesen und beschrieben werden.



Es handelt sich dabei um Textdateien, in welchem sich in jeder Zeile ein Datensatz befindet. In der ersten Zeile befindet sich der **optionale** Header mit den Namen der Spalten. Ein Datensatz besteht aus verschiedenen Elementen, die durch einen Separator (hier das Komma) getrennt sind.

Mit dem Komma als Separator (Trennzeichen) gab es aber in der Vergangenheit Probleme. Je nach eingestellter Landessprache wird bei der Ausgabe einer Fließkommazahl ein Punkt oder ein Komma gesetzt. Wird beim Einlesen dann das Komma als Separator verwendet, so würde z.B. 3,2 (statt 3.2) als zwei Elemente interpretiert werden. **Um dieses Problem zu umgehen werden heutzutage das Semikolon oder ein Tab-Zeichen (\t) als Separator verwendet.**

## 2. csv-Dateien

16

### Beispiel für die Verarbeitung

Zeilenweises Einlesen von Datei wie bisher.

```
fgets(acLine, 255, pRMessage);

// without Error-Handling
pcToken = strtok_s(acLine, ",", &pcTokenNext);
printf("%s\n", pcToken);
sStudent.iMatrNo = atoi(pcToken);

pcToken = strtok_s(NULL, ",", &pcTokenNext);
printf("%s\n", pcToken);
strncpy_s(sStudent.acLastName, 40, pcToken, 39);

pcToken = strtok_s(NULL, ",", &pcTokenNext);
printf("%s\n", pcToken);
strncpy_s(sStudent.acFirstName, 40, pcToken, 39);

pcToken = strtok_s(NULL, ",", &pcTokenNext);
printf("%s\n", pcToken);
sStudent.fAverageMarks = (float)atof(pcToken);
```

```
struct Student
{
    int iMatrNo;
    char acLastName[40];
    char acFirstName[40];
    float fAverageMarks;
};
typedef struct Student sStudent_t;
typedef sStudent_t* psStudent_t;
```

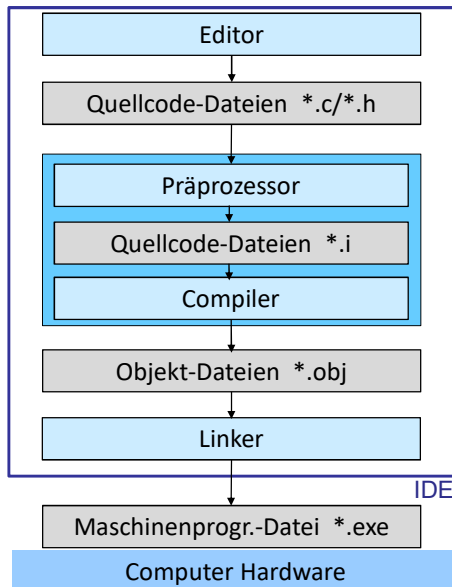
Ja nach „Ziel-Datentyp“ müssen die einzelnen Tokens (char\*) noch umgewandelt werden.

Details zu den Funktionen strtok\_s, atoi und atof in einem späteren Kapitel.



### 3. Präprozessor

17



#### Prinzip

Synonym für **Präprozessor: Präcompiler**

Alle Befehle (genauer: **Direktiven**) für den Präprozessor beginnen mit # (HashTag). Intern wandelt der Präprozessor \*.c und \*.h Dateien in \*.i-Dateien um und übersetzt diese dann zu \*.obj-Dateien. Die temporären \*.i-Dateien (nach [LUH17] auch \*.pp-Dateien) werden durch den Compiler per Default meist gelöscht.

Soll jedoch nur eine \*.i-Datei erzeugt werden ist in Visual Studio die folgende Einstellung vorzunehmen:

Project -> Properties -> C/C++ -

>Preprocessor->Process to a File ->Yes (/P)

Die \*.i-Datei erscheint dann im Debug-Verzeichnis des Projektes. Eine \*.obj-Datei wird dann aber nicht generiert.

### 3. Präprozessor

18

#### #include-Direktive

```

#include <stdio.h>
#include "AppDeclarations.h"

int main(void)
{
    sStudent_t sStudent = { 199999,
                           "Huber",
                           "Max",
                           2.7f };
    printf("Name of Student: %s",
          sStudent.acLastName);

    return 0;
}
  
```

Mit #include wird die komplette Headerdatei eingefügt. Header der Standardbibliotheken mit <> und eigene Header mit "".

```

// more content
_CRT_STDIO_INLINE int __CRTDECL
printf(_In_z_
       _Printf_format_string_
       char const* const _Format,
       ...);
// more content
stdio.h
  
```

```

#pragma once

struct Student
{
    int iMatrNo;
    char acLastName[40];
    char acFirstName[40];
    float fAverageMarks;
};
typedef struct Student sStudent_t;
typedef sStudent_t* psStudent_t;
AppDeclarations.h
  
```

### 3. Präprozessor

19

#### Übersicht #define-Direktive

#define gibt es in **drei verschiedenen Varianten**

- Als **Definition**, die später mit Steueranweisungen (#ifdef) abgefragt werden kann. Anwendung findet dies z.B. in der **bedingten Compilierung** oder als **Include-Guard**.

```
#define STUDENTS_AI
```

- Als **symbolische Konstante** (siehe KE1). Der Präprozessor ersetzt diese dann in der \*.i-Datei durch eine literale Konstante

```
#define STRING_LENGTH_NAME 40
```

- Als **Funktionsmakro**

```
#define MAX(a,b) ((a)>(b) ? (a) : (b))
```

 Siehe hierzu 4. in dieser KE

Die **Namen der Defines** sind **immer groß** zu schreiben, um diese besser von Funktionen, Variablen und konstanten Variablen zu unterscheiden (C-Coding Styleguide P3).

### 3. Präprozessor

20

#### #define-Direktive als Definition für bedingte Compilierung

```
#define STUDENTS_AI
```



```
#ifdef STUDENTS_AI
    printf("Hallo Studierende von AI\n");
#else
    printf("Hallo Studierende von EI, EIplus, EI3nat, MKA, MKplus\n");
#endif
```

Je nachdem, ob STUDENTS\_AI definiert ist, wird ein anderer Code erzeugt. Die IDE MSVS gibt hier auch visuelle Unterstützung. Die „inaktive“ Code erscheint im Editor heller.

In diesem Zusammenhang werden dann auch die Direktiven **#ifdef**, **#else**, **#endif**, **#ifndef**, **#elif** und **#if** eingesetzt.

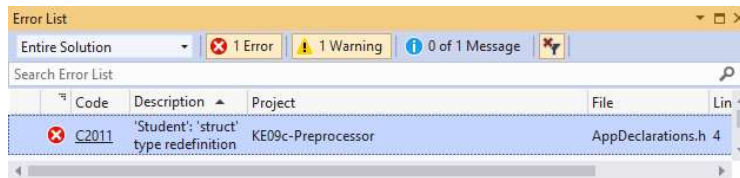
In Softwaresystemen, die für unterschiedliche Plattformen (Betriebssysteme oder Hardware) eingesetzt werden, findet sich häufig bedingte Compilierung. Dadurch wird der Programmcode komplexer. Bei der Cyclomatic Complexity wird dies auch zu berücksichtigt. #ifdef zählt dort auch wie eine „normale“ Bedingung.

### 3. Präprozessor

21

#### #define-Direktive als Definition für Include-Guard

Problemstellung: Jede C-Datei wird separat übersetzt und erst danach werden alle vom Compiler erzeugten \*.obj-Dateien miteinander verlinkt. Eine C-Datei darf nur einmal eine Headerdatei einbinden (C-Coding Styleguide P1), ansonsten erscheint z.B. der folgende Error.



Abhilfe schafft ein Include-Guard der dies verhindert (z.B. für AppDeclarations.h)! Es wird der Dateiname als Name für das define verwendet. Ein „\_“ kennzeichnet üblicherweise system-nahe Komponenten – zwei „\_“ werden für Komponenten des Betriebssystems verwendet.

```
#ifndef _APPDECLARATIONS_H
#define _APPDECLARATIONS_H

//Content of Header file

#endif // _APPDECLARATIONS_H
```

Einige Compiler-Hersteller wie MS gehen bezüglich Include-Guards einen anderen Weg. Beim Anlegen einer neuen Headerdatei fügt es **#pragma once** ein. Der Effekt ist der gleiche, allerdings funktioniert dies nur für bestimmte Compiler.

### 3. Präprozessor

22

#### #define-Direktive als symbolische Konstante

Problemstellung: Oft wird systemweit ein einheitlicher Wert benötigt. Dies kann mit einer symbolischen Konstante realisiert werden. In allen Fällen, in denen stattdessen eine konstante Variable verwendet werden kann, sollte auf den Einsatz einer symbolischen Konstanten verzichtet werden (C-Coding Styleguide P8).

```
#define PI 3.1415
```

Zu vermeiden!

```
const double cdPi = 3.1415;
```

Besser

Es gibt aber Anwendungsfälle, die nur mit symbolischer Konstante funktionieren (C89).

Beispiel: An verschiedenen Stellen in einem SW-System werden Zeichenketten für Nachnamen verwendet. Diese sollten eine einheitliche Länge haben.

```
#define STRING_LENGTH_NAME 40
```

STRING\_LENGTH\_NAME

```
char acLastName[25] = {0};
```

```
struct Student
{
    int iMatrNo;
    char acLastName[40];
    char acFirstName[40];
    float fAverageMarks;
};
```

### 3. Präprozessor

23

#### Weitere wichtige Direktiven

##### #inline

Neuvergabe von Zeilennummer

##### #

Operator zur Textersetzung

##### ##

Operator zur Verkettung zweier Texte

##### #error

Ausgabe von Fehlern während Compilierung

##### #pragma

Compilerspezifische Schlüsselworte

##### #undef

Definitionen können wieder rückgängig gemacht werden.

### 4. Funktionsmakros und Inlinefunktionen

24

#### Nach- und Vorteile einer Funktion

Der Aufruf einer Funktion hat **zwei Nachteile**, die insbesondere bei kurzen Funktionen (wenige Anweisungen) ins Gewicht fallen:

- Beim Aufruf einer Funktion und beim Rücksprung muss zusätzlicher Code generiert werden. Auf dem Stack müssen Variablen abgelegt werden – ebenso die Rücksprungadresse ist zu sichern.
- Dadurch wird das Programm größer und langsamer.

Die **Vorteile** von Funktionen – gegenüber dem mehrfachen Vorhandensein von Programmcode wurde bereits genannt: Der Programmcode ist nur einmal vorhanden, dadurch

- sind Änderungen nur einmal durchzuführen
- ist dieser übersichtlicher und auch kleiner
- Wird die Wartbarkeit verbessert

Funktionsmakros und Inlinefunktionen sind Sprachelemente von C, welche **die Vorteile von Funktionen nutzen, ohne deren Nachteile zu haben**. Umgangssprachlich ausgedrückt: Diese Sprachelemente sehen aus wie Funktionen im Quellcode, sind aber im Maschinencode dann doch mehrfach ohne Aufruf vorhanden.

## 4. Funktionsmakros und Inlinefunktionen

25

### Funktionsmakro - Prinzip

Ein Funktionsmakro ist ein `#define` mit Parametern. Das erste Leerzeichen nach der Parameterliste (nach dem „`)`“) trennt die Deklaration von der Implementierung des Makros.

**Kein Leerzeichen!**

```
#define NAMEFUNKTIONSMAKRO(Param1, Param2, ...) (Makrobefehle)
```

**Leerzeichen!**

Beispiel:

```
#define POWER3(a) ((a) * (a) * (a))
```

...

```
iY = POWER3(iX);
```

...

```
iY = ((iX) * (iX) * (iX));
```

**\*.c-Datei**

Präprozessor

**\*.i-Datei**

Wie auch bei den symbolischen Konstanten ist dies eine **reine Textersetzung**.

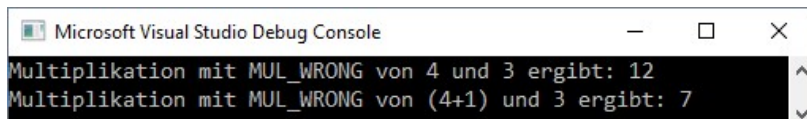
## 4. Funktionsmakros und Inlinefunktionen

26

### Funktionsmakro – Fehleranfälligkeit (1)

```
#define MUL_WRONG(a,b) a*b
```

```
iX = 4;  
iY = 3;  
iZ = MUL_WRONG(iX, iY);  
printf("Multiplikation mit MUL_WRONG von %d und %d ergibt: %d\n",  
       iX, iY, iZ);  
iZ = MUL_WRONG(iX+1, iY);  
printf("Multiplikation mit MUL_WRONG von (%d+1) und %d ergibt: %d\n",  
       iX, iY, iZ);
```



```
Microsoft Visual Studio Debug Console  
Multiplikation mit MUL_WRONG von 4 und 3 ergibt: 12  
Multiplikation mit MUL_WRONG von (4+1) und 3 ergibt: 7
```

**Alle Makroparameter müssen geklammert werden.**

Abhilfe für das aufgezeigte Problem:

```
#define MUL(a,b) (a)*(b)
```

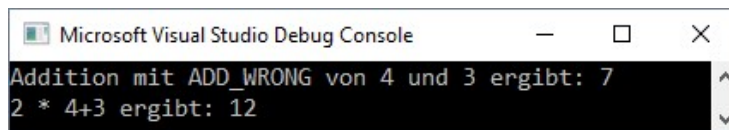
## 4. Funktionsmakros und Inlinefunktionen

27

### Funktionsmakro – Fehleranfälligkeit (2)

```
#define ADD_WRONG(a,b) a+b
```

```
iX = 4;  
iY = 3;  
iZ = ADD_WRONG(iX, iY);  
printf("Addition mit ADD_WRONG von %d und %d ergibt: %d\n",  
      iX, iY, iZ);  
iZ = 2 * ADD_WRONG(iX + 1, iY);  
printf("2 * %d+%d ergibt: %d\n", iX, iY, iZ);
```



**Das gesamte Makro muss geklammert werden.**

Abhilfe für das aufgezeigte Problem:

```
#define ADD(a,b) (a+b)
```

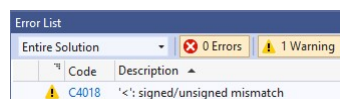
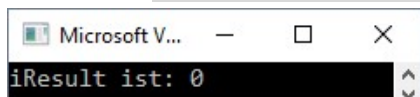
## 4. Funktionsmakros und Inlinefunktionen

28

### Funktionsmakro – Fehleranfälligkeit (3)

```
#define SMALLER10_WRONG(Value1,Value2) (((Value1) + 10) < (Value2)) ? (1) : (0)
```

```
iValue1 = -42;  
uiValue2 = 42;  
iResult = SMALLER10_WRONG(iValue1, uiValue2);  
printf("iResult ist: %d\n", iResult);
```



MS C-Compiler erkennt hier ein Mismatch. Wenn Treat Warnings As Errors auf Yes(/WX) steht, dann wird kein Executable generiert. Nicht jeder C-Compiler erkennt hier ein Mismatch.

**Jeder Makroparameter ist zu casten, ebenso der Rückgabewert**

```
#define SMALLER10(Value1,Value2) (int)((((int)(Value1) + 10) < (int)(Value2)) ? (1) : (0))
```

## 4. Funktionsmakros und Inlinefunktionen

29

### Beispiel Funktionsmakro: `_countof` aus `stdlib.h`

```
int aiArray[] = {1, 2, 3, 4};
```

Das Array `aiArray` enthält vier Elemente. Durch Weglassen der Größe mit `[]` bestimmt der Compiler dessen Größe.

```
printf("Number of Elements in aiArray: %d\n", _countof(aiArray));
```

Für die Bestimmung der Größe steht in `stdlib.h` (kein ANSI-Standard, MS-Erweiterung) das Funktionsmakro `_countof` zur Verfügung. Entgegen unserem C-Coding Styleguide ist das Funktionsmakro nicht in Großbuchstaben geschrieben, d.h. auf den ersten Blick ist dies nicht als Funktionsmakro erkennbar (könnte auch eine Funktion sein). Ein Underscore weist auf eine systemnahe Funktion hin.

In `stdlib.h` wird `_countof` erst als `__crt_countof` deklariert (symbolische Konstante). `__crt_countof` steht dann als Funktionsmakro zur Verfügung:

```
#define _countof __crt_countof
```

```
#define __crt_countof(_Array) (sizeof(_Array) / sizeof(_Array[0]))
```

16

4

## 4. Funktionsmakros und Inlinefunktionen

30

### Zusammenfassung Funktionsmakros

Funktionsmakros werden als Sprachelement häufig eingesetzt. Sie sind jedoch recht fehleranfällig, was die vorherigen Beispiele aufzeigten. Selbst mit Casts sind diese **nicht typsicher**.

Funktionsmakros haben sehr viele **Seiteneffekte**, an die oft nicht gedacht wird.

Ebenso muss nochmals hervorgehoben werden, dass Funktionsmakros, bzw. `#define`, nur reine Textersetzungen sind. Daher lassen sich Funktionsmakros auch **schwer zu debuggen**.

Mit C99 hat ein Sprachelement Einzug in C gefunden, welches die Leistungsfähigkeit von Funktionsmakros hat, allerdings die obigen Nachteile vermeidet: **Inline-Funktionen**

Für gewisse Anwendungsbereiche haben aber Makros noch ihre Daseinsberechtigung.

## 4. Funktionsmakros und Inlinefunktionen

31

### Inlinefunktionen

Mit dem Keyword inline wird dem Compiler mitgeteilt, dass der Code dieser Funktion im Maschinencode den Aufruf ersetzt. Dies ist allerdings nur als Empfehlung anzusehen – der Compiler entscheidet selbst ob dies so umgesetzt wird oder nicht.

**Vorteil:** Der Aufruf und Rücksprung wird eingespart.

**Nachteil:** Code wird größer, da an allen Stellen, wo der Funktionsaufruf stand, der Programmcode der Funktion sich jetzt befindet.

Sinnvoll ist das Inlining somit bei kurzen Funktionen!

#### Funktionsdeklaration

```
inline char GetMaxChar(char cC1, char cC2);
```

#### Funktionsdefinition

```
inline char GetMaxChar(char cC1, char cC2)
{
    char cRet;

    if (cC1 > cC2)
    {
        cRet = cC1;
    }
    else
    {
        cRet = cC2;
    }

    return cRet;
}
```

## 4. Funktionsmakros und Inlinefunktionen

32

### Inlinefunktionen – Microsoft C-Compiler Option

```
cChar = GetMaxChar('1', 'X');
printf("cChar ist: %c\n", cChar);
```

Project -> Properties -> C/C++ -> Optimization ->

**Disassemblerfenster** (Debug->Windows->Disassembly)

```
cChar = GetMaxChar('1', 'X');
00DD5223 6A 58          push     58h
00DD5225 6A 31          push     31h
00DD5227 E8 5D C1 FF FF  call     _GetMaxChar (0DD1389h)
00DD522C 83 C4 08       add     esp,8
00DD522F 88 45 B3       mov     byte ptr [cChar],al
printf("cChar ist: %c\n", cChar);
```

**Funktionsaufruf**

```
cChar = GetMaxChar('1', 'X');
009853C3 B8 31 00 00 00  mov     eax,31h
009853C8 83 F8 58       cmp     eax,58h
009853CB 7E 06          jle     main+173h (09853D3h)
009853CD C6 45 A7 31     mov     byte ptr [ebp-59h],31h
009853D1 EB 04          jmp     main+177h (09853D7h)
009853D3 C6 45 A7 58     mov     byte ptr [ebp-59h],58h
009853D7 8A 4D A7       mov     cl,byte ptr [ebp-59h]
009853DA 88 4D B3       mov     byte ptr [cChar],cl
printf("cChar ist: %c\n", cChar);
```

**Inlining**

->Disabled (/Od)

->Inline Function Expansion->

Only \_inline (/Ob1)  
**-Inlining-**

oder

Any Suitable (/Ob2)  
**-Compiler entscheidet-**



## Behandelte Schlüsselwörter in KE 9

Schlüsselwörter C89:

<del>auto</del> ✓	do ✓	<del>goto</del> ✓	signed ✓	unsigned ✓
break ✓	double ✓	if ✓	sizeof ✓	void ✓✓
case ✓	else ✓	int ✓	static ✓	volatile ✓
char ✓	enum ✓	long ✓	struct ✓	while ✓
const ✓	extern ✓	register ✓	switch ✓	
<del>continue</del> ✓	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			