1 Grundlagen

Siehe auch: https://de.wikipedia.org/wiki/C-Präprozessor
Siehe auch: info/cpp.info Inhalt der GCC-Distribution
gcc-arm-none-eabi-10-2020-q4-major\share\doc\gcc-arm-none-eabi\info\cpp.info

Der Präprozessor ist ein von C-Syntax unabhängiger Sprachsyntax. Beim Übersetzungsprozess einer C-Datei wird dieser als zweiter Prozess (nach dem Entfernen der Kommentare) ausgeführt und das Ergebnis dieser Übersetzung dem eigentlichen C-Compiler vorgelegt.
Der Präprozessor erzeugt dabei keinen ausführbaren Code (also keine Maschinensprachebefehle wie der C-Compiler) sondern entspricht im Wesentlichen einer Textersetzung, wie Sie diese aus ihren Texteditor kennen. Prinzipiell könnte der C-Präprozessor auch als Präprozessor für andere Sprachen genutzt werden.
Das Resultat des Übersetzungsprozesses kann mittels des Compilerschalters 'gcc -E' angezeigt werden (im Compiler Explorer dazu -E im Feld 'Compiler options …' eintragen.) Dieser stoppt den Übersetzungsprozess nach Durchlauf des Präprozessors und gibt den erzeugten C-Code auf der Standardausgabe aus. Vorsicht, bei vielen Include-Anweisungen ist die Ausgabe schnell mal mehrere Tausend Zeilen lang.

Der Syntax des Präprozessors weicht deutlich vom C-Syntax ab. Wesentliche Eigenschaften des Syntax sind:
- Kein Semikolon zum 'Abschluss' eines Befehls
- Zeilenfortführung in nächster Zeile mit Backslash '\' als letztes Zeichen in der Zeile (entsprechend String)
- Viele Präprozessorbefehle beginnen mit einen '#', welchem beliebig viele Leerzeichen vorangestellt sein können.

Die Befehle lassen sich in folgende Kategorien unterteilen:
- Include-Anweisung
- Object-Like Makos
- Function-Like Makros
- Bedingte Übersetzung
- Sonstiges

1.1 Include-Anweisung

Entspricht einer Textersetzung, wobei der Ausdruck '#include <….>' durch den Inhalt der dazugehörigen Datei ersetzt wird.
Die Include-Anweisung kann jede Datei inkludieren. Damit der anschließende Compiler keine Fehler meldet, sollte diese jedoch C-Syntax enthalten. Typischerweise werden die inkludierten Files Header-Files genannt und haben die Dateiendung .h oder .hpp.
Die includierten Files werden ebenfalls vom Präprozessor geparst, so dass z.B. hier enthaltene Include-Anweisungen ebenfalls aufgelöst werden (Vorsicht: Gefahr einer Endlosschleife und Gefahr von Doppelten Include von identischen Dateien)

Syntax

```
#include <h-char-sequence >
#include "y-char-sequence"
#include Preprocessor-Tokens    //Standard-C
```

Beispiel für Include

```
#include <stdio.h>
#include "class.h"
#define includefile1 <stdint.h>
#define includefile2 "main.h"
#include includefile1
//Kein Semikolon am Ende der Zeile. Dieses würde nicht ersetzt werden
//sondern im resultierenden Code erhalten bleiben
```

Suchpfade
"y-char-sequence": Suchpfad für die einzubindende Datei ist das aktuelle Arbeitsverzeichnis, in welche, die Ausgangsdatei liegt. Über relative Pfadangaben können Dateien in 'Nachbarschaft' und über absolute Pfadangabe Dateien von 'überall' inkludiert werden.

```
#include "test.h"
#include "verzeichnis/hallo.h"
#include "c:\user\xyz\test.h"
#include "../lib/test.h"
```

Vorsicht, Windows nutzt '\' und Unix nutzt '/' zur Pfadtrennung (wobei der GCC beide unabhängig vom Betriebssystem akzeptiert)
<h-char-sequence>: Nutzung des vom Compiler vorgegebenen Suchpfades zum Suchen nach der angegebenen Datei. Im Compiler-Suchpfade sind unter anderen die Verzeichnisse der Header-Dateien der Standard-C-Library enthalten. Über relative Pfadangaben können Dateien in 'Nachbarschaft' zum Suchpfad inkludiert werden

```
#include <stdio.h>
#include <sys/stat.h>
```

Mittels `'gcc -Ixxx'` kann der Suchpfade um die Pfadangabe xxx ergänzt werden, so dass z.B. die Header-Dateien von Librarys nicht über absolute Pfadangaben sondern direkt angesprochen werden können.
Die Liste der vom Compiler genutzten Suchpfade kann mittels 'cpp -v' oder 'cpp -v /dev/null -o /dev/null' ausgedruckt werden

Include-Dateien werden u.A. für folgenden Funktionalitäten genutzt:

- Deklarationen von Funktionen, die in einer C-Datei geschrieben sind und von anderen C-Dateien genutzt werden sollen (entspricht der 'Public' Anweisung für Funktionen und Attribute):

Class.h

```
int class_set(int val,int attr);
int class_init(void);
```

Class.c

```
int class_set(int val,int attr)
{
   …
}
int class_init(void)
{
   …
}
```

Func1.c

```
#include "class.h"

int func1_init(void)
{
   ...
   class_init();
   …
   class_set(4,1);
}
```

Func2.c

```
#include "class.h"

int func2_init(void)
{
  ..
  class_init();
  …
  class_set(47,11);
}
```

- Definition von projektweiten Konstanten und Datentypen, welche von allen C-Dateien genutzt werden können:

Global.h

```
#define DNS_ADDR "141.41.1.150"

#define ERROR(text, arg... ) \
        (fflush(stdout),\
         fprintf(stderr,\
         "\e[31m%s() Error:\e[30m "\
         text"'\n",__func__,##arg) )

#define MODE_TemperaturOnly  1
#define MODE_DruckOnly       2
#define MODE_AttitudeOnly    3
#define MODE_TemperaturDruck 4
#define MODE  MODE_AttidueOnly

typedef struct {
   int mode;
   int debuglevel;
   …
} global_t;
```

- Damit die Funktionen von Library Dateien genutzt werden können, sind in Header-Dateien auch die Deklarationen der in den Librarys enthalten Funktionen und Datentypen enthalten (Diese werden dann zumeist über #include <> inkludiert und die Suchpfade zu diesen werden beim Compileraufruf gesetzt)

Stdio.h

```
…
extern int fprintf (FILE *__restrict __stream,
                    const char *__restrict __format, ...);

extern int printf (const char *__restrict __format, ...);

extern int sprintf (char *__restrict __s,
                   const char *__restrict __format, ...) __attribute__ ((__nothrow__));

extern int vfprintf (FILE *__restrict __s,
                     const char *__restrict __format,  __gnuc_va_list __arg);
….
```

Die Prototypen, die Datentypen und die Konstanten der Standard-C Library sind in diverse Header-Dateien verteilt: https://de.wikipedia.org/wiki/C-Standard-Bibliothek

Bei unachtsamer Nutzung von include Anweisung können gewollt/ungewollt:
1) Die identische Header-Datei mehrmals includiert werden (direkt oder auch indirekt)

Datei.h	

`#include <stdio.h>`
`//'indirekt'`
`int var; //*1)`
`typedef unsigned int UI;`
`#define HALLO 1`	

	Main.c

	`#include <stdio.h>`
	`#include "datei.h"`
	`#include "datei.h"`
	`//direkt`

2) Eine Header-Datei in unterschiedlichen Dateien inkludiert werden

Datei.h		

`int var; //*2)`
`typedef unsigned int UI;`
`#define HALLO 1`		

	Func1.c	Func2.c

	`#include`
	`"datei.h"`
	`.…`
		`#include`
		`"datei.h"`
		`.…`

Dementsprechend sind bezüglich des Inhaltes von Header-Dateien die Regelwerke von C zu beachten:

- Definitionen nur einmal
  - Wenn eine Header-Datei mehrmals inkludiert wird (siehe 1)) oben), würde z.B. eine Variable oder eine Funktion folglich doppelt definiert werden, was zu einem Compilerfehler führt
  - Wenn eine Header-Datei in unterschiedlichen Dateien inkludiert wird (siehe 2)) oben), dann würde in jeder C-Datei eine unabhängige Variable/Funktion angelegt werden, so dass der Linker im Anschluss aufgrund der doppelten Namensgebung den Linkvorgang abbricht.

  --> Daher gilt, in einer Header-Daei sind keine Definitionen erlaubt
- Beliebig viele Deklarationen (sofern diese natürlich vom identischen Datentyp sind)
- Datentypen dürfen nicht doppelt definiert werden (siehe *1) oben)
- Makros dürfen nicht doppelt definiert werden (siehe *1) oben)

Include Wächter
Identische Header können mehrmals inkludiert werden. Zur Vermeidung von doppelten Deklarationen, Datentypen, Markos werden Include-Wächter verwendet, die händisch in jeder Header-Datei zu schreiben sind:

```
#ifndef _Projektname_Dateiname_Dateiendung_INCLUDED_
#define _Projektname_Dateiname_Dateiendung_INCLUDED_
//Eigentlicher Inhalt der Header-Datei
#endif
```

Alternativ gibt es eine Compileranweisung (Nicht C-Standard, wird jedoch von vielen Compilern unterstützt), welche komfortabler ist und doppele Namen vermeidet! (überliest nachfolgende Include Anweisungen)
#pragma once

Sonstiges (nur zur Info)

- Include lädt einzig die Deklarationen in die aktuelle C-Datei ein. Die dazugehörigen Definitionen sind beim Linker Durchlauf, z.B. durch Librarys separat beizufügen.
- C++ benutzt ein Name Mangling System, so dass auf deren Funktionen und Variablen nicht direkt zugegriffen werden kann. Dieses Name Mangling kann wie folgt deaktiviert werden, so dass CPP-Funktionen aus C aufgerufen werden können

```
extern "c" void test(void);
#ifdef __cplusplus
extern "C" {
#endif
void test(void);
#ifdef __cplusplus
}
#endif
```

- Zum Inkludieren von Standard-C Header Dateien in C++ muss dem Dateienamen ein c vorangestellt werden #include <cstring>. Die .h Endung entfällt
- Mit jeder Include Anweisung erhöht sich die Zeitdauer für einen Compiledurchlauf. Einerseits muss jede zu inkludierende Datei von der Festplatte geladen werden und andererseits vergrößert sich mit jeder Datei der zu übersetzende Code. In diesen Sinn gilt, Weniger Include Anweisungen ist Mehr. Also gerne mal regelmäßig die Include-Anweisungen durchgehen und überlegen, ob diese in der Tat noch benötigt werden

1.2 Object-like-Makros

Syntax: #define name sequence-of-tokens$_{opt}$

Entspricht einer Textersetzung, wobei das Wort name im **folgenden** Code durch sequence-of-tokens ersetzt wird. Innerhalb von Kommentare und Strings erfolgt keine Textersetzung.
Die bedingte Übersetzung (siehe später) prüft oftmals nur ab, ob ein Makro definiert ist (#ifdef oder #if defined()). Der 'zugewiesene' Wert 'sequence-of-tokens' ist dabei nicht von Interesse, so dass sequence-of-tokens optional ist / entfallen kann.
Wie auch Variablen und Funktionsnamen dürfen Makros nicht doppelt definiert werden. Jedoch kann mit #undef ein zuvor definiertes Makro gelöscht werden, so dass dann mit #define eine neue Zuweisung erfolgen kann.
Tipp: gcc -E zur Darstellung der Übersetzung nutzen
Beispiele:

```
#define   ROT1 5
#define   ROT2
    #   define ROT3     8  //Leerzeichen werden Eliminiert
#define   ROT4 /*Test */ "Hello World"
#define  ROT5 5;
    int a=ROT5,b=ROT5+6;   //???

#define  ROT6 1,"hallo",7
  char str[]="ROT6"
  Struct {int a; char str[7]; int b} var={ROT6};

#define  else
     if(1==2) printf("hallo"); else printf("welt");

#define ADDAB a+b
   int  a,b,c=ADDAB;

#define  ROT1  //KO da Mehrfachdefinition verboten
#undef   ROT1  //für begrenzte Gültigkeit oder Überschreibung

#define ROT7 BACKSLASH \
#define ROT8  8                     //??

#define XYZ  9
enum {XYZ=9};                       //??
```

Ergänzend zu den selbst definierten Makros schreibt der C-Syntax folgende vordefinierten Makros vor, welche vom Compiler durch die entsprechenden Gegebenheiten ausgetauscht werden:

```
__LINE__    --> Datentyp: int     Wird ersetzt durch die aktuelle Zeilennummer, in welcher das
                                  Makro steht
__FILE__    --> Datentyp: char *  Wird ersetzt durch den Dateinamen der Datei
__DATE__    --> Datentyp: char *  Wird ersetzt durch das aktuelle Datum zum Compilezeitpunkt
__TIME__    --> Datentyp: char *  Wird ersetzt durch die aktuelle Uhrzeit zum Compilezeitpunkt
__STDC__    --> Datentyp: int     Wert=1, wenn Compiler Standard-C konform ist
__STDC_VERSION__ --> Datentyp: int YYYYMM  Jahr und Monat der C-Version
                                           (199409  für C89 199901 für C99)
__func__ (ab C99) --> Datentyp: char *  wird ersetzt durch einen Zeiger auf einen String,
                      in welchen der aktuelle Funktiosname enthalten ist
```

Beispiel:

```
#define MELDUNG(text) fprintf( stderr, "Datei [%s], Zeile %d: %s\n" ,__FILE__, __LINE__, text )
```

Weiterhin gibt es Makros, die vom Compiler in Abhängigkeit des Betriebssystems, der Rechenbreite, des Prozessors und vielen anderen Bedingungen gesetzt werden. Diese dienen unter anderem dafür, plattformunabhängigen Code zu schreiben:

```
__GNUC__              --> Defined, wenn es sich um einen GNU Compiler handelt
__INCLUDE_LEVEL__     --> Zahl, welche den aktuellen Inklude-Level angibt
__CHAR_UNSIGNED__     --> Defined, wenn Char vom Datentyp unsigned ist
__TIMESTAMP__         --> String mit dem letzten Änderungsdatum der Datei
```

Und viele andere mehr, z.B.

```
__WIN32    __unix__  __linux__  __i386__ __CYGWIN32__
```

Die vom Compiler vordefinierten Makros können mit nachfolgender Anweisung ausgegeben/angezeigt werden

```
>> cpp -dM -E -xc /dev/null
```

Ergänzend zur #define Anweisung können Makros auch über den Compileraufruf gesetzt wer

```
gcc -Dxxxx[=definition]              //für Object-Like Makros
gcc -Dxxx[(arglist)=Definition]      //für Function-Like Makros
```

Über das Makro NDEBUG wird i.d.R. gesagt, dass der erzeugte Code ein 'Release' Code, also für die Auslieferung ist. Mittels bedingter Übersetzung können auf Grundlage dieses Makros z.B. Debug-Ausgaben unterbunden werden. Das Makro wird entweder über die Entwicklungsumgebung gesetzt oder ist im makefile enthalten, so dass im Release Mode ergänzende Compileroptimierungen eingeschaltet werden können.

1.3 Function-like-Makros

Syntax: #define name (identifier-List$_{opt}$) sequence-of-tokens$_{opt}$

identifier-List$_{Opt}$ ist eine kommaseparierte Liste, die auch leer sein kann.
Zwischen Namen und '(' ist kein Leereichen erlaubt (andernfalls handelt es sich dann um ein objekt-like-Makro und die öffnende Klammer gehört zu sequence-of-tokens).
Entspricht einer 'doppelte' Textersetzung, wobei zunächst nach dem Wort name gesucht und dieses durch sequence-of-tokens$_{OPT}$ ersetzt. Innerhalb von sequence-of-tokens$_{OPT}$ erfolgt nun eine zweite Ersetzung, wobei die Identifier-List durch die tatsächlichen 'Parameter' ersetzt werden.

```
#define ADD(a,b) a+b
int var=ADD(7,8);
```

1.) Ersetzungsschritt: int var=a+b(7,8);
2.) Ersetzungsschritt: int var=7+b(8);
3.) Ersetzungsschritt: int var=7+8;

Beispiele:

```
#define   ADD(a,b) a+b
int   a=1  ,b=2  ,c=ADD(a,b);
float x=1.0,y=2.0,z=ADD(x,y);
char  o=5  ,p=8  ,q=ADD(o,p)*ADD(o,p);   //Inhalt von q?
q=ADD( ,8);   //Inhalt von q? (bitte mit gcc -E selbst herausfinden)
```

```
q=ADD( , );   //Inhalt von q? (bitte mit gcc -E selbst herausfinden)
q=ADD( );     //Inhalt von q? (bitte mit gcc -E selbst herausfinden)


//Aufgrund obiger Probleme empfiehlt sich, 'sequence-of-tokens'
//zu klammern!
#define ADD2(a,b)  (a+b)

#define MAX(a,b)   (a>b?a:b)
int a=7,b=8,c=MAX(a++,b++);   //Inhalt von c?
                              //(bitte selbst herausfinden)
```

Das Problem hierbei ist, dass der Übergabeparameter nicht vor der Ausführung der Funktion, sondern während der Ausführung der Funktion ausgewertet werden (Call-by-Name/Call-by-macro expansion).

```
//Aufgrund obigen Problems empfiehlt sich, 'sequence-of-tokens'
//in einen eigenen Block-Scope auszuführen, so dass hier 'temporäre'
//Variablen genutzt werden können. Die Auswertung der Übergabeparameter
//erfolgt dann entsprechend dem Call-By-Value Ansatz mit dem Makroaufruf
#define MAX(a, b) { __typeof__ (a) _a = (a); \
                    __typeof__ (b) _b = (b); \
                    _a > _b ? _a : _b; }

if(a>b)
MAX(a++,b++);
else                     //Compilerfehler!
printf("Hallo");

//Das 'Semikolon' Problem kann wie folgt umgangen werden
#define MAX1(a, b) ({ __typeof__ (a) _a = (a); \
                  __typeof__ (b) _b = (b); \
                      _a > _b ? _a : _b; })

//Alternativ könnte auch die Funktionalität in einer do while() Schleife
//gekapselt werden, wobei dann kein Rückgabewert möglich ist
#define XYZ(a,b)  do  { … } while(0)
```

Function-Like Makros sind ein mächtiges, aber auch fehleranfälliges Werkzeug. Es empfiehlt sich folglich:

- Ausreichend Klammern (sowohl () als auch ({ }))
- **Möglichst Makronamen in GROSSBUCHSTABEN schreiben, so dass beim Lesen des Source-Codes klar ist, dass hier ein Makro und keine Variable/Funktion aufgerufen wird.**
- Innerhalb von Makros den Komma-Operator zum Trennen von Anweisungen, anstatt das Semikolon zu verwenden
- Bei Makroaufrufen sind Argumente mit den Operatoren ++ und -- sowie Funktionen und Zuweisungen als Argumente zu vermeiden, da diese durch eventuelle Mehrfachauswertung zu unerwünschten Seiteneffekten oder sogar undefiniertem Code führen können.

Function-Like Makros bieten sich in folgenden Situationen an:

- Funktionalitäten unabhängig vom Datentyp bereitstellen

```
#define MAX(a,b)  ({  /* siehe oben */ })
#define FOREACH(iterator,array)   for(typeof(array[0]) *iterator; ….
```

- Als Alternative zu einen Funktionsaufruf (entspricht einer inline Funktion)

```
#define read_SCL(port)  (*AT91C_PIOA_PDSR & i2c_mask[port][1])
#define set_SCL(port)   (*AT91C_PIOA_SODR = i2c_mask[port][1])
#define clear_SCL(port) (*AT91C_PIOA_CODR = i2c_mask[port][1])
```

- Lesbareren und dennoch schnellen Code zu erzeugen

```
struct vl {
    struct vl *next;
};
struct vl *liste = NULL;
//Unleserlich
strcpy(((char *)&liste[1]), "key1");
strcpy(((char *)&liste[1]+strlen(((char *)&liste[1]))+1),"value1");

#define VL_KEY(vl)    ((char *)&vl[1])
#define VL_VALUE(vl)  ((char *)&vl[1]+strlen(VL_KEY(vl))+1)
//Besser
strcpy(VL_KEY(liste),  "key2");
strcpy(VL_VALUE(liste),"value2");
```

1.4 Bedingte Übersetzung

Die bedingte Übersetzung erlaubt es, Textbereiche ein- resp. auszublenden (d.h. Text-Bereich aus dem Source-Code zu entfernen)
Syntax:

```
#if const-expression-1
    group-of-lines-1  //beliebige Anzahl an Zeilen, die bei gültiger Bedingungen
                      //eingeblendet, andernfalls ausgeblendet werden
#elif const-expression-2
    group-of-lines-2
   …
#else const-expression-x
   group-of-lines-x
#endif

//--> Verschachtelung möglich
```

Const-expression:
Integer-Konstante oder Makro/Ausdruck (==, >= ! & | && || + - << >> …) welcher **zur Compilezeit** zu einer Integer-Konstante aufgelöst wird 0->False <>0->True.

Beispiel:

```
#define A  1
#if 1
#if A
#if A == 1
#if (A+1) == 1
```

Ein nicht definiertes Makro wird zu 0 ersetzt. Ein definiertes Marko ohne Wertzuweisung wird durch nichts ersetzt, so dass ein Vergleich mit einer Interkonstanten fehlerhaft ist.

Beispiel:

```
#define A 1
#define B
#if A==1
#if B==0    //Fehler
#if C==0
```

Ergänzende Operatoren:

- 'defined name' resp. 'defined (name)' wird zu 1 aufgelöst, wenn name als Makro definiert wurde (egal, ob und welcher Inhalt zugewiesen wurde), andernfalls 0
- Kurzform
  - #ifdef name entspricht #if defined name
  - #ifndef name entspricht #if !defined name

Beispiel

```
#define A
#if    defined A
#if ! defined (B)
#ifdef A
#ifndef B
```

Anwendung
- Debug/Release (Im Debug-Mode zusätzliche Debug-Ausgaben aktivieren)
  ```
  #ifdef NDEBUG
  ```
- Host-System Abhängigkeiten nutzen (GCC,CLANG,Mircosoft / Windows/Linux)
  ```
  #ifdef __unix__
  ```
- Traget-Unabhängigkeiten (für unterschiedliche Prozessor, Ressourcenausstattung)
- Derivate-Steuerung (Low-Cost / High-Cost)
  ```
  #define VERSION_CHROM_V90  90
  #define VERSIION_CHROM_V89 89
  #if VERSION == VERSION_CHROM_V90 && defined __unix__
  ```

1.5 Error/Warning Anweisung

Erzeugung einer Compiler Warning/Fehler-Meldung mit dem Inhalt des Token (proprocesor-token muss daher kein String sein)
Syntax:
```
#error   preprocessor-tokens
#warning preprocessor-Tokens   //Kein C-Standard, sondern
                               //Bestandteil von gcc
```

Anwendung
- Zur Überprüfung, ob Makros 'richtig' gesetzt wurden

```
#define BUF_SIZE
#if (BUF_SIZE % 256) != 0
    #error   Bufsize muss ein vielfaches von 256 betragen
#endif
#ifndef __unix__
  #error   Windows ist doof!
#endif
```

1.6 Pragma

Anweisung an den Compiler zum Aktivieren/Deaktivieren bestimmter Funktionalitäten
Syntax:
`#pragma preprocessor-tokens`

Die preprocessor-Tokens sind Compiler-Abhängig

Beispiel:

```
//CompilerOptimierung einschalten
#pragma GCC push_options
#pragma GCC optimize ("-O3")
//Einschaltung der max. Compiler Optimierung für diese eine Funktion
void ws2812_send(void)
{
}
#pragma GCC pop_options

//Anweisung an den Compiler, dass nachfolgende Schleife
//parallelisiert werden kann
#pragma omp parallel for reduction(+:c)
 for (int i = 0; i < length; ++i)
     c += a[i]*b[i];
```

1.7 Stringized-Operator

Wird einem Parameter im Ersatztext eines Function-Like-Makros ein # vorangestellt, so wird bei der Ersetzung (durch den Präprozessor) das Argument durch Einschließen in doppelte Hochkommata in eine Zeichenkette umgewandelt (stringized).

Beispiel: Folgendes Programm gibt 'String' aus, nicht 'hallo':

```
#define STR(X) #X
char string[] = "hallo";
puts( STR( string ) );
```

Anwendung

- Zum Aufbau einer eigenen Symboltabelle, in welcher der Name der Variablen und die Adresse der Variablen enthalten ist

```
struct var {int *adr; char name[100];};
#define VAR(x) {&x,#x}
int a,b,c;
struct var var1[]={VAR(a),VAR(b),VAR(c)};
```

1.8 Verkettung von Makroparametern / Token Merging

Der Verkettungsoperator ## erlaubt es, zwei Makroparameter innerhalb eines Function-like-Makros zu einem zu verschmelzen:

Beispiel:

```
#define GLUE(X,Y) X ## Y
printf( "%d\n", GLUE(2, 34) );   //Gibt 234 als Zahl zurück

#define TEMP(i)  temp##i
TEMP(1) = TEMP(2 +k)   +x;     //-->temp1=temp2+k+x;

#define PRIVATE(member) private_##member
struct class {
    int PRIVATE(xyz);
};
```