Preprozessor

- Der Proprozessor ist ein dem eigentlichen C-Compiler vorgelagertes Programm.
- Er liest und erzeugt c-Quelltext.
- Er erkennt spezielle Quelltextkonstrukte und ersetzt diese durch andere.
- Im Ergebnis entsteht wieder ein c-Quelltext, der dann vom eigentlichen c-Compiler weiterverarbeitet wird.
- Mit `gcc -E x.c > x.e` wird gezielt nur der Preprozessor ausgeführt, x.e enthält den preprozessierten Quelltext.

trigraphs

- Trigraphs sind sog. Zeichenersatz-folgen.
- Um sie zu nutzen, muss die Compi-leroption **-trigraphs** angegeben werden.
- Der Präprozessor ersetzt die tri-graphs durch das jeweilige Zei-chen.

Trigraph	Zeichen
??=	#
??/	\
??'	^
??(	[
??)	]
??<	{
??>	}
??!	|
??-	~

Zeilenverlängerung

- Prinzipiell ist die Länge von Programmzeilen in c unererheblich.
- Bestimmte Konstrukte (Macros) sind aber an Zeilen gebunden.
- Sehr lange Quelltextzeilen sind unübersichtlich.
- Mit Zeilenverlängerung kann man eine Zeile umbrechen. Diesen Umbruch entfernt dann der Preprozessor.
- Die Zeilenverlängerung wird durch einen \ als letztes Zeichen einer Zeile markiert.
- Selbst Zeichenketten können damit auf mehrere Zeilen aufgeteilt werden.

```
1  #include <stdio.h>
2
3  int main()
4  {
5   puts("Spass\
6   beim\
7   Programmieren");
8   return 0;
9  }
```

```
$ ./a.out
Spass beim Programmieren
$
```

#include

- #include ist eine Preprozessoranweisung.
- Die includeanweisung wird aus dem Quelltext entfernt und durch die angegebene Datei ersetzt.
- Meist werden headerfiles mit #include eingebunden.
- Werden die einzubindenden Files in '<' und '>' eingeschlossen, so wird die angegebene Datei laut Voreinstellung in den System-headerfiles gesucht.
- Mit der Compileroption -I können weitere Suchpfade für System-headerfiles hinzugefügt werden.
- Anwendungsspezifische headerfiles werden in '"' geklammert.

Testen Sie :
gcc E sinus.c > sinus.e und schauen suchen Sie in sinus.e nach der mainfunktion.

Symboldefinitionen

- Mit **`#define`** können Symbole definiert und ihnen Werte zugeordnet werden.
- Soche eine Symboldefinition beginnt mit ‚#define und endet mit dem Zeilenende.
- Mittels Zeilenverlängerung kann sich eine Symboldefinition auch über mehrere Zeilen erstrecken.
- Auf das Schlüsselwort define folgt das zu definierende Symbol, dann nach wenigstens einem whitespace der Ersetzungstext.
- Im letzten Beispiel wird len ersetzt durch
  80 + 1

Beispiele:
```
#define TRUE 1
#define FALSE 0
#define OK 0
#define pfirst pnxt
#define plast pprv
#define len 80 + 1
```

Symboldefinitionen

- Symboldefinitionen sind der klassische Weg, in c Symbolische Konstanten zu vereinbaren.
- Das ist vor allem bei mehrfach auftretenden Konstanten interessant, da sie, wenn sie per #define definiert wurden, nur zentral an einer Stelle geändert werden können.
- Dabei ist zu beachten, dass der Precompiler nur Quelltext ersetzt.

Beispiele:

```
#define TRUE 1
#define FALSE 0
#define OK 0
#define pfirst pnxt
#define plast pprv
#define len 80 + 1
#define e1 “usage %s <num>\n“
```

```
char ThreeNames[len*3];
```

Preprozessor

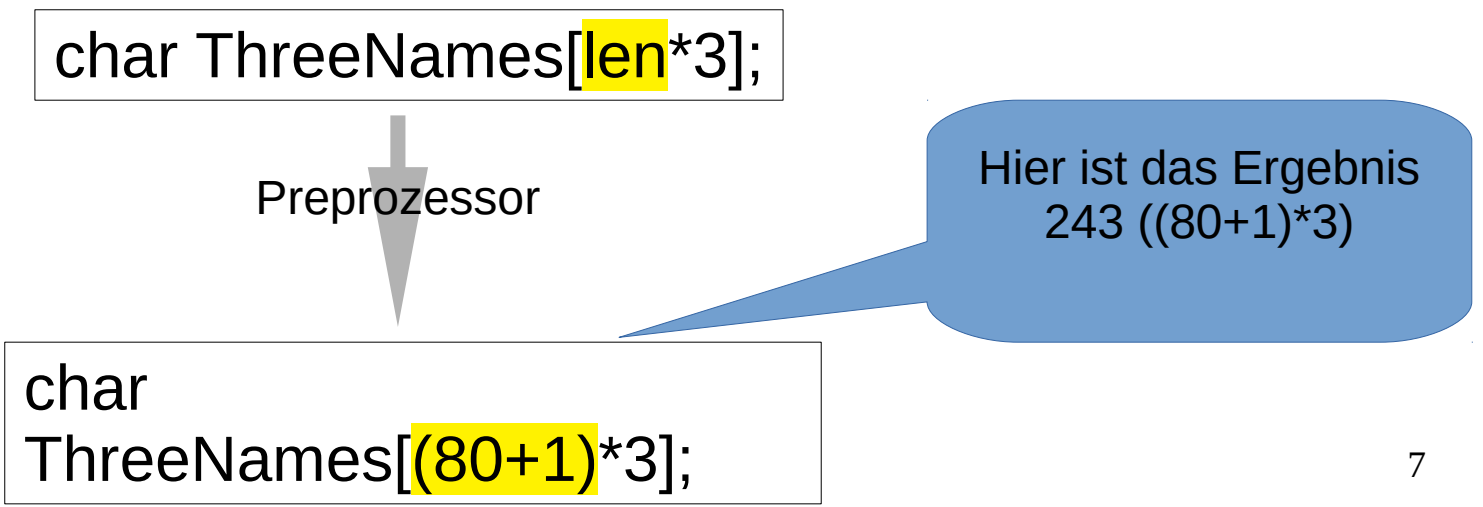
```
char ThreeNames[80+1*3];
```

Das Ergebnis ist nicht
243 ((80+1)*3)
sondern 83 (80+1*3)

Symboldefinitionen

- Deshalb ist es sinnvoll, Klammerungen in dem Ersetzungstext zu setzen, auch wenn das an der Stelle nicht sinnvoll erscheint.
  `#define len (80 + 1)`

Beispiele:
#define TRUE 1
#define FALSE 0
#define OK 0
#define pfirst pnxt
#define plast pprv
#define len (80 + 1)



Symboldefinitionen

- Symboldefinitionen können auch durch den Compiler erzeugt werden.
- Mit der Compileroption -D SYMBOL wird das Symbol SYMBOL mit dem Wert 1 definiert und ist im compilierten c- Programm verfügbar.
- Mit -D SYMBOL=Wert Kann ein Symbol mit einem zu übergebenden Wert definiert werden.

Beispiele:
-D DEBUG
-D FCPU=4000000UL
-D _DEVICE_=2

Symboldefinitionen

- Es gibt eine Vielzahl systemabhängiger Symbole, die vordefiniert sind.
- Das Kommando `gcc -dM -E - < /dev/null` bewirkt die Ausgabe der vordefinierten Symbole der aktuellen Plattform.
- Via Dateiumleitung können die Symbole in eine Datei geschrieben werden: `gcc -dM -E - < /dev/null > symbols.txt`.
- Führen Sie das Kommando aus und durchsuhen Sie die Ausgabe nach für Sie interessant scheinenden Informationen.

Bedingte Übersetzung

- Der Preprozessor gestattet, Programmteile in Abhängigkeit von Bedingungen oder der Existenz von Symbolen einzubinden oder auszuschließen.
- Auch dies passiert auf Quelltextebene, Textpassagen werden in die Ausgabe des Preprozessors übernommen oder auch nicht.
- Auf diese Weise können Plattformabhängigkeiten oder spezielle Konfigurationen eingestellt oder berücksichtigt werden.

Preprozessoranweisungen zur bedingten Übersetzung

- #if <const_expr>
- #if defined <ident>
- #if !defined <ident>
- #ifdef <ident>
- #ifndef <ident>
- #elif <const_expr>
- #else
- #endif

- Bedingte Übersetzung wird immer mit einer **`#if`**-Anweisung eingeleitet und mit **`#endif`** abgeschlossen.
- Es kann meherere **`#elif`**-Zweige und ein **`#else`** geben.
- Zwischen diesen Preprozesoranweisungen befindet sich Code, der nur unter den gennten Bedingungen in den zu compilierenden Quelltext übernommen wird.

Beispiel zur bedingten Übersetzung

```c
#include <stdlib.h>

int main()
{
#ifdef linux
    system("clear");
#else
    system("cls");
#endif
    return 0;
}
```

- Es wird das Vorhandensein des Symbols **`linux`** geprüft.
- **`#ifdef linux`** ist eine Kurzform von **`#if defined linux`**
- Ist das Symbol definert, wird **`system("clear");`** übernommen, ansonsten **`system("cls");`**
- **`system`** führt ein shellkommando aus einem c-Programm heraus aus, hier den Befehl zum Löschen der Konsole.

Beispiel zur bedingten Übersetzung

```
1231  # 3 "condCompiling1.c"
1232  int main()
1233  {
1234
1235      system("clear");
1236
1237
1238
1239      return 0;
1240  }
1241
```

```
1230  # 3 "condCompiling1.c"
1231  int main()
1232  {
1233
1234
1235
1236      system("cls");
1237
1238      return 0;
1239  }
```

- Nebenstehende Abbildung zeigt die letzten Zeilen des precompilierten Quelltextes
  `gcc -E condCompiling.c >condCompiling.e`
- Erinnerung: nachfolgendes Kommado liefert eine Auflistung vordefinierter Symbole
  `gcc -dM -E - < /dev/null`
- Mit der gcc-Option -U kann ein vordefiniertes Symbol gelöscht werden.
  `gcc -U linux -E condCompiling.c >condCompiling.e`
  liefert untenstehenden Quelltext.
- Mit #elif können weitere Systeme getestet werden.

Beispiel zur bedingten Übersetzung

```
1  /* list.h */
2  #ifndef _LIST_H_
3  #define _LIST_H_
4  #define OK 1
5  #define FAIL 0
6  /*---------------------------------------------------------------------------*/
7  /* Prototypen fuer die Funktionen                                            */
8
9  struct tlst;»   »       »       // Forewarddeclaration
10 typedef struct tlst tList;
11
12 tList * CreateList(void);                          /* erzeuge leere Liste */
13 int     DeleteList(tList* pList);                  /* loesche leere Liste */
14
15 int    InsertBehind  (tList* pList, void *pItemIns);/* fuege ein hinter %  */
16 int    InsertBefore  (tList* pList, void *pItemIns);/* fuege ein vor    %  */
17 int    InsertHead    (tList* pList, void *pItemIns);/* fuege vorn ein      */
18 int    InsertTail    (tList* pList, void *pItemIns);/* fuege hinten ein    */
19 int    RemoveItem    (tList* pList);                /* loesche %           */
20
21 void* GetSelected    (tList* pList);                /* gib aktuellen DS    */
22 void* GetFirst       (tList* pList);                /* gib ersten DS       */
23 void* GetLast        (tList* pList);                /* gib letzten DS      */
24 void* GetNext        (tList* pList);                /* gib naechsten DS    */
25 void* GetPrev        (tList* pList);                /* gib vorigen DS      */
26 void* GetIndexed     (tList* pList,int Idx);        /* gib DS lt. Index    */
27
28 void* addItemToList (tList* pList,
29                      void * pItem,
30                      int(*fcmp)(void*pItList,void*pItNew));
31
32 /* % steht fuer aktuellen Satz                                               */
33 #endif
```

- Bedingte Übersetzung wird auch verwendet, um ungewollte Mehrfachincludes eines Headerfiles zu verhindern.
- Nur wenn das Symbol **`_LIST_H_`** nicht definiert ist, wird der Text bis **`#endif`** übernommen.
- Ist das Symbol bereits definiert, wurde das Headerfile list.h bereits includiert.

Macros

- Macros kann man sich zunächst als parametrisierte Symboldefinitionen vorstellen.
- Macros werden mit **`#define`** vereinbart.
- Die Parameter werden in runde Klammern, unmittelbar auf den Macronamen folgend (ohne whitespace!!) durch ihren Namen angegeben.
- Da alles auf Quelltestersatz basiert, gibt es keine getypten Parameter.
- Die Macrodefinition endet am Zeilenende (Zeilenverlängerung ist möglich) ohne ein Semikolon.

Macros

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define H(a,b) sqrt (a*a + b*b)
6
7   int main()
8   {
9     double t1=2.0;
10    double t2=3.0;
11    double c;
12    c=H(t1,t2);
13    printf("c: %lf\n",c);
14    return 0;
15  }
```

- In Zeile 5 wird ein Macro zur Berechnung der Hypotenuse eines rechtwinkligen Dreiecks definiert.
- In Zeile 12 wird das Macro benutzt.
- Dabei wird der parametrisierte Text des Macros anstelle von H(t1,t2) eingesetzt.

Macros

```
 7  int main()
 8  {
 9    double t1=2.0;
10    double t2=3.0;
11    double c;
12    c=H(t1+1,t2+1);
13    printf("c: %lf\n",c);
14    return 0;
15  }
```

```
2756  # 7 "macro1.c"
2757  int main()
2758  {
2759    double t1=2.0;
2760    double t2=3.0;
2761    double c;
2762    c=sqrt (t1+1*t1+1 + t2+1*t2+1);
2763    printf("c: %lf\n",c);
2764    return 0;
2765  }
```

- Nebenstehend ist eine Variante des Programmes dargestellt. Als Parameter werden t1+1 und t2+1 übergeben.
- Bei Funktionen ist das unproblematisch.
- Was bei einem Macro passiert, zeigt das untere Listing, Zeile 2762.
- Zu beachten: Auch hier findet nur Quelltextersatz statt.
- Im Ergebnis wird etwas völlig Falsches berechnet.

Macros

```
5   #define H(a,b) sqrt ((a)*(a) + (b)*(b))
6
7   int main()
8   {
9     double t1=2.0;
10    double t2=3.0;
11    double c;
12    c=H(t1+1,t2+1);
13    printf("c: %lf\n",c);
14    return 0;
15  }
```

```
2756  # 7 "macro1.c"
2757  int main()
2758  {
2759    double t1=2.0;
2760    double t2=3.0;
2761    double c;
2762    c=sqrt ((t1+1)*(t1+1) + (t2+1)*(t2+1));
2763    printf("c: %lf\n",c);
2764    return 0;
2765  }
```

- Abhilfe schafft hier scheinbar sinnlose Klammerung in der Macrodefinition (Zeile 5)
- In Zeile 5, oberes Listing sind a und b geklammert.
- Das Ergebnis (unten Zeile 2762) ist nun wieder korrekt.
- **Zusätzliche Klammerungen sind in Macros sinnvoll.**

Macros

```
5   #define MAX(x1,x2) (x1)>(x2)?(x1):(x2)
6   int main()
7   {
8     double x1=2.0;
9     double x2=3.0;
10    double max;
11    max=MAX(x1++,x2++);
12    printf("max: %lf\nx1: %lf, x1: %lf\n",max,x1,x2);
13    return 0;
14  }
```

```
$ ./a.out
max: 4.000000
x1: 3.000000, x1: 5.000000
$
```

```
2755  # 6 "macro2.c"
2756  int main()
2757  {
2758    double x1=2.0;
2759    double x2=3.0;
2760    double max;
2761    max=(x1++)>(x2++)?(x1++):(x2++);
2762    printf("max: %lf\nx1: %lf, x1: %lf\n",max,x1,x2);
2763    return 0;
2764  }
```

- Ein weiteres Beispiel zeigt die Auswirkung von Seiteneffekten, wie sie bei den Icrement- und Decrement-operationen (**++ , −−**) auftreten.
- Wäre MAX eine Funktion, hätten wir für max: 3.0, für x1: 3.0 und x2: 4.0 erwartet.
- Die Präprozessorausgabe zeigt auch hier, wie das unerwartete Ergebnis zu Stande gekommen ist.
- **Keine Seiteneffekte, wenn Macros im Spiel sind!**

Debugausgaben steuern

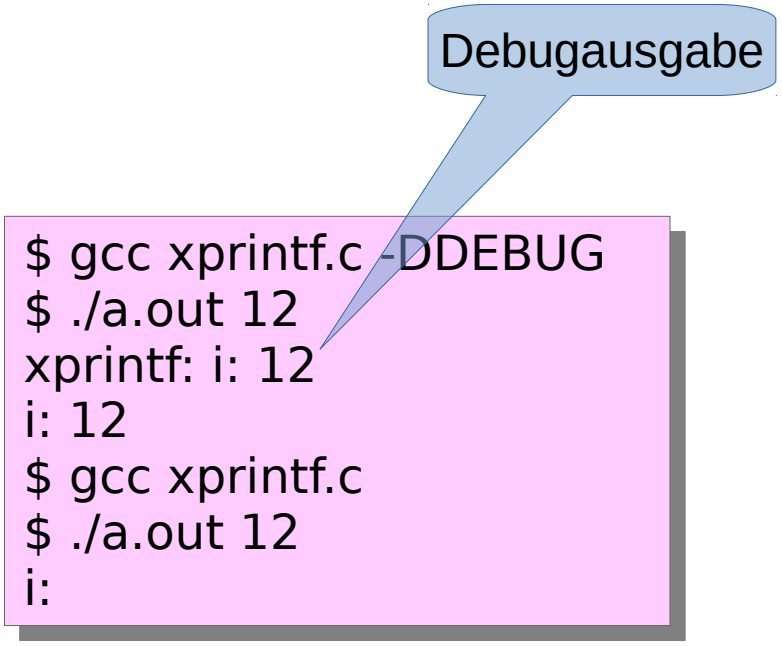
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifdef DEBUG
#define xprintf(x) printf x
#else
#define xprintf(x) /**/
#endif

int main(int argc, char*argv[])
{
  int i=atoi(argv[1]);
  xprintf(("xprintf: i: %d\n",i));
  printf("i: %d\n",i);
  return 0;
}
```

- Im Beispiel wird in Abhängigkeit des Symbols DEBUG ein Macro xprintf definiert.
- Der Parameter umfasst die gesamte Parameterliste von printf, einschließlich der Klammern.
- Wird das Programm mit -D DEBUG compiliert, erfolgen Debugausgaben, wird es ohne -D DEBUG compiliert, fallen die Debugausgaben weg.

Debugausgaben steuern



- Debugausgaben sind zusätzliche Ausgaben auf die Konsole, um Fehlern auf die Spur zu kommen.
- Nicht immer ist der Einsatz eines Debuggers zielführend und sinnvoll.
- Durch die Verwendung eines solchen Macros kann die Ausgabe der hilfreichen Debugmessages mit dem Compilerkommando gesteuert werden.
- gcc -DDEBUG Debugausgaben werden generiert
- gcc ohne -DDEBUG Debugausgaben werden nicht erzeugt.

Macros - Stringverkettung

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  #define CONCAT(x1,x2) "Das ist ein Text mit "#x1" und "#x2
6  int main()
7  {
8    printf("%s\n",CONCAT(Jux,Spass));
9    return 0;
10 }
```

- Folgen Parameter auf ein # so werden sie mit den umgebenden Strings zu einem neuen String verknüpft.
- Im Beispiel werden die beiden Wörter Jux und Spass in den Text als Parameter x1 und x2 in Zeile 5 eingesetzt.

```
$ ./a.out
Das ist ein Text mit Jux und Spass
```

Macros - Stringverkettung

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

#define CONCAT(x1,x2) "Das ist ein Text mit "#x1" und "#x2
int main()
{
  printf("%s\n",CONCAT(Jux,Spass));
  return 0;
}
```

```
$ gcc macro3.c
$ ./a.out
Das ist ein Text mit Jux und Spass
$
```

- Das Ergebnis lässt sich wieder mit gcc -E macro3.c > macro3.e visualisieren (unten).
- Gcc macro3.c liefert das auführbare a.out.

```c
# 6 "macro3.c"
int main()
{
  printf("%s\n","Das ist ein Text mit ""Jux"" und ""Spass");
  return 0;
}
```

Macros Symbolverkettung

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


#define ConcatSymb(x,y) x##y
int main()
{
  if(ConcatSymb(lin,ux)==1)
        puts ("Linux defined");
  return 0;
}
```

- Im Beispiel werden lin und ux durch das Macro ConcatSymb zu einem neuen Symbol zusammengesetzt (linux). Da das Symbol linux mit 1 definiert ist, ergibt sich der Vergleich 1==1 → true.

Vordefinierte Symbole des gcc

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



int main()
{
  printf ("%-16s: %s\n","__FILE__",__FILE__);
  printf ("%-16s: %d\n","__LINE__",__LINE__);
  printf ("%-16s: %s\n","__DATE__",__DATE__);
  printf ("%-16s: %s\n","__TIME__",__TIME__);
  printf ("%-16s: %d\n","__STDC__",__STDC__);
  printf ("%-16s: %ld\n","__STDC_VERSION__",__STDC_VERSION__);
  printf ("%-16s: %d\n","__STDC_HOSTED__",__STDC_HOSTED__);
  return 0;
}
```

```
$ ./a.out
__FILE__        : macro5.c
__LINE__        : 10
__DATE__        : Jan  2 2021
__TIME__        : 18:31:49
__STDC__        : 1
__STDC_VERSION__: 201112
__STDC_HOSTED__ : 1
```

- Auch der gcc stellt Symbole bereit.
- Die Angaben beziehen sich auf den Übersetzungszeitpunkt und das gerade zu compilierende c-File.
- Hosted 1 bedeutet, es gibt ein Betriebssystem. Bei einachen Microcontrollern erhält man 0.