Modellierung und Programmierung

Dr. Martin Riplinger

16.1.2013

C-Präprozessor: Einfügungen

Standard-Header

- **Syntax**: `#include <systemheader.h>`
- Fügt den Text der Headerdatei `systemheader.h` an der Stelle im Code ein, an welcher der `#include`-Befehl steht.
- Die Datei wird im Standard-Header-Pfad des Systems gesucht. Bei Unix-Systemen ist dies üblicherweise `/usr/include`.

  Beispiel: `#include <math.h>` bindet die Datei `/usr/include/math.h` ein.

  Bei Headern in Unterordnern muss dieser mit angegeben werden, z. B. `#include <sys/time.h>` für den Header `/usr/include/sys/time.h`.

Lokale (eigene) Header

- **Syntax**: `#include "lokaler_header.h"`
- Bindet die Datei `lokaler_header.h` ein, die im **aktuellen Verzeichnis** liegt. Will man einen Header aus einem anderen Pfad inkludieren, muss dem Compiler der Pfad mitgeteilt werden.

  Beispiel: Zum Einbinden des Headers `matrix.h` im Unterordner `meine_header` wird
  - im Quellcode die Zeile `#include "matrix.h"` eingefügt.
  - der Compiler mit der Option `-Imeine_header` aufgerufen.

  Die Option `-IOrdner` macht die Header in `Ordner` für den Compiler sichtbar.

C-Präprozessor: Einfügungen – Beispiel `matrix.h`

Headerdatei `matrix.h` im Unterordner `meine_header` (Ausschnitt):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Legt Speicher fuer eine neue Matrix an und liefert einen Zeiger auf die
5  // erste Zeile; bei Misserfolg NULL
6  double **neueMatrix(int n_zeilen, int n_spalten);
7
8  // Gibt die Matrix am Bildschirm aus
9  void zeigeMatrix(double **matrix, int n_zeilen, int n_spalten);
```

Programmcode `matrix_test.c`:

```
1  #include "matrix.h"
2
3  int main(void)
4  {
5    int n = 4, m = 7;
6    double **H = neueMatrixAusBinDatei("Matrix.bin", n, m);
7    zeigeMatrix(H, n, m);
8    loescheMatrix(H, n);
9
10   return 0;
11 } // Hiernach die Funktionsdefinitionen!!
```

Compiler-Aufruf: `gcc -o matrix_test matrix_test.c -Imeine_header`

C-Präprozessor: Makros ohne Parameter I

Syntax: `#define MAKRO_OHNE_PARAMETER`

- Definiert ein Makro mit dem Namen `MAKRO_OHNE_PARAMETER`
- **Namenskonvention**: Makros werden **immer** durchgehend mit **Großbuchtstaben** benannt.
- Mit `#ifdef MAKRO_OHNE_PARAMETER` bzw. `#ifndef MAKRO_OHNE_PARAMETER` kann abgefragt werden, ob das Makro definiert bzw. nicht definiert ist.

  Syntax:

  ```
  #ifdef MAKRO
  // Weitere Direktiven, z. B. #include
  #endif
  ```

  Verwendungsbeispiel: Verhindern, dass ein Header mehrfach eingebunden wird.

  ```
  #ifndef __MEIN_HEADER_H__
  #define __MEIN_HEADER_H__

  // Weitere Direktiven, Funktionsdeklarationen etc.

  #endif
  ```

C-Präprozessor: Makros ohne Parameter II

Syntax: `#define MAKRO Ersetzungstext`

- Bewirkt, dass überall im Quellcode, wo `MAKRO` steht, der `Ersetzungstext` eingesetzt wird.
- **Beispiele**:
  - `#define ANTWORT 42`

    Alle Vorkommen von `ANTWORT` im Quellcode wird **textuell** durch `42` ersetzt. Auf diese Weise brauchen für Konstanten keine Variablen deklariert zu werden.
  - `#define ADRESSE long unsigned int`

    In der Folge kann `ADRESSE` wie ein Datentyp verwendet werden. So lassen sich intuitive Namen für Datentypen vergeben.
- **Generelle Verwendung**: Vergabe von **sprechenden Namen** für Konstanten, Datentypen, ...
  - Vermeidung von „magische Zahlen“ (`PUFFERGROESSE` hat mehr Aussagekraft als `1024`).
  - Durch Änderung der Definition lassen sich alle Vorkommen der Konstanten schnell und unkompliziert anpassen.

  Folge: Besser lesbarer und leichter zu pflegender Code

C-Präprozessor: Makros mit Parametern

Syntax: `#define MAKRO(Parameterliste) Ersetzungstext`

- Im Code lässt sich `MAKRO` wie eine Funktion aufrufen.
- **Achtung**: Es wird nicht überprüft, ob die Argumente passende Typen haben.
- Generell umfassen Makros nur eine einzige Zeile. Lange Zeilen lassen sich mit einem Backslash \ am Ende umbrechen.
- **Beispiel 1**: `#define MAX(X,Y) X > Y ? X : Y`

  Berechnet das Maximum von zwei Argumenten
- **Beispiel 2**: `#define TAUSCHE(X,Y) int z = X; X = Y; Y = z;`

  Vertauscht die Werte von zwei `int`-Argumenten.

C-Präprozessor: Makros mit variabler Anzahl von Parametern

Syntax: `#define MAKRO(Parameterliste,...) Ersetzungstext`

- Werden in Verbindung mit Funktionen verwendet, die mit einer variablen Anzahl von Argumenten aufgerufen werden können, z. B. `fprintf`.
- Das oben definierte Makro muss mit **mindestens** so vielen Parametern aufgerufen werden wie die Parameterliste lang ist.
- Auf die über diese Anzahl hinausgehenden Parameter, die anstelle der Punkte `...` übergeben werden, kann mit dem Makro `__VA_ARGS__` zugegriffen werden.
- **Beispiel**:

```
#define EPRINTF(_fmt_str, ...) \
        fprintf(stderr, _fmt_str, ##__VA_ARGS__)
```

  Der Aufruf `EPRINTF("i = %d\n", i);` wird beispielsweise ersetzt durch `fprintf(stderr, "i = %d\n", i);`
- Die doppelte Raute `##` vor `__VA_ARGS__` bewirkt hier speziell, dass das Komma nach dem Formatstring wegfällt, falls `__VA_ARGS__` leer ist. Andernfalls ergäbe sich ein Syntaxfehler.

  Beispiel: `EPRINTF("Hallo!\n");` wird hier ersetzt zum syntaktisch korrekten `fprintf(stderr, "Hallo!\n");` Ohne die Doppelraute würde die Ersetzung `fprintf(stderr, "Hallo!\n", );` lauten, was einen Syntaxfehler darstellt.

C-Präprozessor: Fallstricke

- Argumente sollten im Ersetzungstext **immer geklammert werden**, da sonst zusammengesetzte Ausdrücke als Argument eventuell falsch ausgewertet werden.

  Beispiel: `#define ZWEIMAL(X) 2 * X`

  Der Aufruf `ZWEIMAL(4 + 2)` wird aufgelöst zu `2 * 4 + 2`, im Ergebnis 10.

  Das Makro `#define ZWEIMAL(X) 2 * (X)` liefert das korrekte Ergebnis:

  `2*(4 + 2)`=12.

- Formeln sollten im Ersetzungstext **immer geklammert werden**, da sonst zusammengesetzte Ausdrücke als Argument eventuell falsch ausgewertet werden.

  Beispiel: `#define MAX(X,Y) (X) > (Y) ? (X) : (Y)`

  Der Aufruf `2*MAX(3,5)` wird aufgelöst zu `2 * (3) > (5) ? (3) : (5)`, im Ergebnis 3. Das Makro `#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))` liefert das korrekte Ergebnis: `2*((3) > (4) ? (3) : (5))`=10.

C-Präprozessor: Fallstricke

- Anweisungs**folgen** sind **in geschweifte Klammern** zu setzen.

  Beispiel 1: `#define ABBRUCH_VOID printf("Abbruch!\n"); return;`

  Der Code `if(fehler) ABBRUCH_VOID;` resultiert im aufgelösten Code `if(fehler) printf("Abbruch!\n"); return;` Dadurch wird **immer** das `return`-Statement ausgeführt!

  Mit `#define ABBRUCH_VOID {printf("Abbruch!\n"); return;}`

  tritt dieses Problem nicht auf.

  Beispiel 2: `#define TAUSCHE(X,Y) int z = X; X = Y; Y = z;`

  Falls eine Variable z bereits deklariert wurde, beschwert sich der Compiler!

  Die Definition `#define TAUSCHE(X,Y) {int z = X; X = Y; Y = z;}` behebt diesen Konflikt, da die Deklaration im neuen Anweisungsblock übergeordnete Variablen bis zum Ende des Blocks überdecken kann.

C-Präprozessor: Fallstricke

- Ausdrücke mit **Nebeneffekten** oder Aufrufe von **rechenzeitintensiven** Funktionen sollten in Makros vermieden werden, da sie eventuell mehrfach ausgewertet werden.

  Beispiel 1: `char c = MAX(fgetc(stdin), 'a');` wird aufgelöst zu

  `char c = ((fgetc(stdin)) > ('a')) ? (fgetc(stdin)) : 'a';`
  Falls das erste eingelesene Zeichen (als Zahl) größer ist als `'a'`, so wird ein zweites Zeichen eingelesen.

  Beispiel 2: `int i = 5, j = MAX(i++, 2);` wird aufgelöst zu

  `int i = 5, j = ((i++) > (2)) ? (i++) : (2);`
  Danach gilt nicht wie erwartet `i =` und `j =` , sondern `i =` und `j =` .

  Beispiel 3: `double x = 1.0; double y = MAX(viel_zu_rechnen(x), 0.0);`

  Hier wird unter Umständen die Funktion `viel_zu_rechnen` zweimal ausgewertet, was zu einer doppelt so langen Laufzeit führt!

C-Präprozessor: Vor- und Nachteile von Makros gegenüber Funktionen

Vorteile:

- Makros werden zur Compilezeit ausgewertet und sind in vielen Fällen schneller.
- Viele Makros können universell eingesetzt werden (z. B. MAX für alle vergleichbaren Datentypen).

Nachteile:

- Zu viele oder zu lange Makros lassen den Codeumfang anwachsen und ergeben unter Umständen große und langsame Programme. Zudem ist der Speicherbedarf größer als bei Funktionen.
- Mehrfache Auswertung von Ausdrücken kann unerwünschte Konsequenzen nach sich ziehen.
- Es gibt keine Möglichkeit zu prüfen, ob sinnvolle Datentypen als Parameter verwendet werden.
- Die Fehlerquellen im Umgang mit Makros sind vielfältig.
- Der Versuch, Funktionspointer als Zeiger auf ein Makro zu verwenden, scheitert an einem Syntaxfehler. Dies ist vor allem dann problematisch, wenn nicht klar ist, ob ein Name für eine Funktion oder ein Makro steht.

Fazit: Makros mit Parametern als Ersatz für Funktionen eignen sich für **einfache** Aufgaben in **geschwindigkeitskritischen** Bereichen.

C-Präprozessor: bedingte Ersetzung

Konditionale Direktiven:

```
#if Bedingung1
// Direktiven / Code
#elif Bedingung
// Weitere Direktiven / Code
#else
// Alternative Direktiven / Code
#endif
```

Prinzipiell funktioniert dieses Konstrukt wie C-Statements mit `if` - `else if` - `else` (`#elif` ist eine Kurzform für „else if“). Als Bedingungen können **konstante** Zahlen, andere Makros sowie arithmetische und logische Ausdrücke verwendet werden.

```
#ifdef MAKRO // bzw. #ifndef
// Weitere Direktiven
#endif
```

Direktiven werden ausgeführt, falls `MAKRO` (nicht) definiert wurde.

Wichtig: `#if` und `#ifdef` müssen **immer** durch ein `#endif` abgeschlossen werden.

Definition rückgängig machen:

```
#undef MAKRO
```

C-Präprozessor: bedingte Ersetzung – Beispiel

Header `debug.h`:

```
1  #ifndef __DEBUG_H__
2  #define __DEBUG_H__
3
4  #include <stdio.h>
5
6  #ifdef DEBUG_AN
7  #define DEBUG_AUSGABE(_fmt_string, ...) \
8    fprintf(stderr, "[Datei %s, Zeile %d] " _fmt_string, \
9    __FILE__, __LINE__, ##__VA_ARGS__)
10
11 #else
12 #define DEBUG_AUSGABE(_fmt_string, ...) // definiert als "nichts"
13
14 #endif // DEBUG_AN
15 #endif // __DEBUG_H__
```

Programmcode `debug_test.c`:

```
1  #include "debug.h"
2
3  int main(void)
4  {
5    int i = 42;
6    DEBUG_AUSGABE("Hallo Welt!\n");
7    DEBUG_AUSGABE("i = %d\n", i);
8
9    return 0;
10 }
```

C-Präprozessor: bedingte Ersetzung – Beispiel

Codeanalyse Header:

```
#ifdef DEBUG_AN
#define DEBUG_AUSGABE(_fmt_string, ...) \
  fprintf(stderr, "[Datei %s, Zeile %d]: " _fmt_string, \
  __FILE__, __LINE__, ##__VA_ARGS__)

#else
#define DEBUG_AUSGABE(_fmt_string, ...) // definiert als "nichts"

#endif // DEBUG_AN
```

- Falls `DEBUG_AN` definiert wurde, nimmt der Präprozessor die obere Version des Makros `DEBUG_AUSGABE`, andernfalls die untere leere Version.
- Die hintereinander geschriebenen Strings `"[Datei %s, Zeile %d]: "` und `_fmt_string` werden automatisch zu einem zusammengefügt (*konkateniert*). Dies gilt generell für **konstante** Strings in C.
- `__FILE__` und `__LINE__` sind vordefinierte Makros, die vom Präprozessor durch den **Dateinamen** (als String) und die **Codezeile** (als Ganzzahlkonstante) ersetzt werden. Aus diesem Grund sind dafür die Platzhalter `%s` bzw. `%d` vorgesehen.
- Wurde `DEBUG_AN` nicht definiert, so ersetzt der Präprozessor alle Aufrufe von `DEBUG_AUSGABE` durch **nichts**. In diesem Fall wäre beispielsweise `DEBUG_AUSGABE("Hallo!\n");` ein leeres Statement (`;`).

C-Präprozessor: bedingte Ersetzung – Beispiel

Codeanalyse Hauptprogramm:

```
int i = 42;
DEBUG_AUSGABE("Hallo Welt!\n");
DEBUG_AUSGABE("i = %d\n", i);
```

Compileraufruf 1: `gcc -o debug_test debug_test.c`

- Das Makro `DEBUG_AN` ist **nicht** definiert, daher erfolgt keine Ausgabe.

Compileraufruf 2: `gcc -o debug_test debug_test.c -DDEBUG_AN`

- Durch die Option `-DDEBUG_AN` wird das Makro `DEBUG_AN` definiert, und der Präprozessor verwendet die „gehaltvolle“ Version von `DEBUG_AUSGABE`.
- Das zweite `DEBUG_AUSGABE`-Statement in obigem Code beispielsweise wird wie erwartet durch `fprintf(stderr, "[Datei %s, Zeile %d] " "i = %d\n", i);` ersetzt.
- Generell wirkt die Compileroption `-DMAKRO[=Wert]` wie eine Zeile `#define MAKRO [Wert]` im Code.

  Achtung: Bei Stringkonstanten müssen Zeichen, die normalerweise von der Kommandozeile interpretiert werden, mit einem Backslash „escaped“ werden. Dazu gehören u. a. Anführungszeichen, Hochkommata, Backslash, Dollar, Raute, Klammern, ...

  Beispiel: Um ein Makro `NACHRICHT` mit dem Wert `"Im Westen nichts Neues."` zu definieren, muss die Compileroption `-DNACHRICHT=\"Im Westen nichts Neues.\"` lauten.

C-Präprozessor: mehrstufige Ersetzungen und Zugriff auf Variablennamen

- Es ist möglich, Makros zu definieren, die von anderen Makros abhängen.

  Beispiel: Kurzform eines langen Konstantennamens

  ```
  #define ERDBESCHLEUNIGUNG 9.81
  #define G ERDBESCHLEUNIGUNG
  ```

  - Generell durchläuft der Präprozessor den Code so lange, bis alle Ersetzungen erfolgt sind.
  - Beim ersten Durchlauf werden alle Vorkommen von `G` durch `ERDBESCHLEUNIGUNG` ersetzt. Im zweiten Durchgang erfolgt die endgültige Auflösung zur Konstanten `9.81`.

- Es ist möglich, mit Makros auf den Namen von Variablen zuzugreifen.

  Beispiel: `#define PRINTDOUBLEWITHNAME(X) printf("%s = %.3lf ",#X,X)`

  Im Programm führt der Aufruf

  ```
  double x=1.121,y=2.3; PRINTDOUBLEWITHNAME(x); PRINTDOUBLEWITHNAME(y);
  ```

  dann zu folgender Ausgabe: `x = 1.121 y = 2.300`.