9. Präprozessor

Präprozessor-Anweisungen sind Anweisungen die vor dem eigentlichen Übersetzen des Programms durch den Präprozessor ausgeführt werden. Alle Zeilen die mit dem Zeichen '#' beginnen werden als Präprozessor-Anweisung interpretiert. Vor dem Symbol '#' dürfen beliebig viele Leerzeichen und Tabulatoren stehen. Nach dem Symbol '#' folgt die eigentlichen Präprozessor-Anweisung. Und auch hier gilt: zwischen dem Symbol '#' und der Anweisung dürfen wieder beliebig viele Leerzeichen bzw. Tabulatoren stehen. Diese Anweisungen werden nicht mit einem Semikolon abgeschlossen.

9.1. #include

Diese Anweisung kennen wir schon sehr gut – wir haben sie in jedem Beispiel min. einmal benutzt. Was sie aber genau macht und welche Regeln es dafür gibt, sehen wir erst jetzt.

Es gibt zwei Arten von #include:

Beispiel 1. #include

```
#include <datei>
#include „datei“
```

Die erste Form der Anweisung sucht die einzubindende Datei in einem voreingestelltem Pfad – normalerweise kann man diesen Pfad in der Entwicklungsumgebung einstellen. Alle mit Compiler mitgelieferten Header-Dateien befinden sich dort und werden auf diese Art eigebunden

Die zweite Form sucht die Datei im selben Verzeichnis, in dem sich auch der Quellcode befindet. Wenn sie dort nicht gefunden wird, dann sucht der Präprozessor wieder im voreingestelltem Pfad. Diese Form wird normalerweise für eigene Header-Dateien benutzt.

Bemerkung. Beide Formen lassen sowohl absolute als auch relative Pfadeingaben zu. Beachten Sie aber bitte, dass man in diesem Fall einfache Backslashs benötigt.

Beispiel 2. #include

```
Suche nur im Standard-Include-Pfad
#include <iostream>

Suche im akt. Verzeichnis und im dann im
Standard-Include-Pfad
#include "common.h"

Suche in einem relativen Pfad
#include "..\include\myfile.h"
```

9.2. #define und #undef

Mit der Anweisung #define lassen sich Symbole und Werte (Konstanten) definieren. Z.B.:

Beispiel 3. #define

```
//Definition des Symbols _COMMON_H
#define COMMON_H

// Definition des Symbols DEBUG
#define DEBUG

// Definition des Symbols MAXSIZE mit
// dem Wert 10
#define MAXSIZE 10

// Definition des Symbols ERRTEXT mit
// einem String
#define ERRTEXT "Fehler aufgetreten!\n"
```

Und hier ein vollständiger Beispiel:

Beispiel 4. vollständiger Beispiel

```
// Definition eines Text-Symbols
#define GSTRING "Guten Tag\n"

// Definition zweier int-Symbole
#define MAXSIZE 10
#define LARGESIZE (MAXSIZE*2)

// Variablendefinition
// Die folgende Anweisung wird durch den
// Präprozessor folgendermaßen erweitert:
// char array[10];
char array[MAXSIZE];

// Hauptprogramm
int main ()
{
// Die folgende Anweisung wird erweitert zu:
// cout << "Guten Tag\n";
    cout << GSTRING;
// Aber keine Erweiterung dieser Anweisung da
// das Symbol innerhalb eines Strings steht!
    cout << "GSTRING";
}
```

Wofür man diese Anweisungen und Definitionen braucht, wird gleich klar.

Ntürlich kann man die definierten Symbole und Kontanten auch löschen: mit der #undef-Anweisung.

Beispiel 5. #define und #undef

```
// Symbol DEBUG definieren
#define DEBUG
...
// Symbol wieder löschen
#undef DEBUG
...
```

9.3. #ifdef, #ifndef und #endif

Vielleicht haben Sie sich in der Zwischenzeit gefragt, was Sie mit einem definierten Symbol denn alles anfangen können. Oft es so, dass während der Entwicklung eines Programms verschiedene Meldungen zur Kontrolle des Programmablaufs ausgegeben

werden. In der fertigen, auslieferfähigen Version sollen diese Meldungen dann natürlich nicht mehr erscheinen. Sie könnten jetzt hergehen und alle Meldungen entfernen, müssten dann aber bei einem erneuten Programmtest die Meldungen wieder einbauen. Aber es geht natürlich auch einfacher. Mit der Präprozessor-Anweisung *#ifdef SYMBOL* können Sie abfragen, ob ein bestimmtes Symbol definiert ist oder nicht. Ist das Symbol definiert, so übernimmt der Präprozessor alle Anweisungen die zwischen der *#ifdef* Anweisung und der zu ihr dazugehörigen *#endif* Anweisung stehen. Ist das Symbol dagegen nicht definiert, so werden diese Anweisungen vor dem Compiliervorgang 'entfernt', d.h. der Compiler bekommt die Anweisungen erst gar nicht 'zu Gesicht', und erzeugt somit auch keinen Code dafür. Im Beispiel rechts werden die beiden Ausgaben nur dann in den Code übernommen, wenn das Symbol *DEBUG* definiert ist.

Beispiel 6. #ifdef und #endif

```
#define DEBUG
....
int main ()
{
    ....
#ifdef DEBUG
    cout << "Programmpunkt1" << endl;
#endif
    ....
#ifdef DEBUG
    cout << "Variablenwert" << ....
#endif
    ....
}
```

Außer der *#ifdef SYMBOL* Anweisung gibt es noch die *#ifndef SYMBOL* Anweisung. Sie hat die entgegengesetzte Wirkung wie *#ifdef*, d.h. der zwischen *#ifndef* und *#endif* stehende Code wird nur dann übernommen, wenn das Symbol nicht definiert ist. Diese Anweisung spielt bei den Dateien eine wichtige Rolle, die mittels *#include* eingebunden werden. Sehen Sie sich dazu einmal das Beispiel rechts an. Die Datei *FILE1.H* enthält z.B. beliebige Definitionen und Deklarationen. Einige dieser Definitionen und Deklarationen werden jetzt z.B. auch von der Datei *FILE2.H* benötigt, weshalb *FILE2.H* die Datei *FILE1.H* einbindet. So weit ist alles noch in Ordnung. Sehen wir uns jetzt einmal eine typische Anwendung dazu an. Da die Quellcode-Datei *MAIN.CPP* sowohl Definitionen und Deklarationen aus der Datei *FILE1.H* wie auch aus *FILE2.H* benötigt, wird *MAIN.CPP* auch beide Dateien einbinden. Und schon hätten wir ein 'kleines' Problem. Denn zuerst bindet *MAIN.CPP* die Datei *FILE1.H* ein und fügt dadurch bestimmte Deklarationen bzw. Definitionen ein. Anschließend bindet *MAIN.CPP* nun die Datei *FILE2.H* ein. Da *FILE2.H* selbst aber nochmals die Datei *FILE1.H* einbindet, würden damit die Definitionen/Deklarationen aus *FILE1.H* doppelt in *MAIN.CPP* vorkommen, was dann zu einem Übersetzungsfehler führt. Auch das Vertauschen der *include*-Anweisungen in *MAIN.CPP* bringt keine

Lösung. Vielleicht sagen Sie sich nun: dann binde ich in der Datei *FILE2.H* die Datei *FILE1.H* einfach nicht mehr ein und schon funktionierts. Im Prinzip haben Sie damit auch recht, doch sollten Sie niemals das Einbinden einer Datei vom vorherigen Einbinden einer anderen Datei abhängig machen. Im Beispiel wäre das erfolgreiche Einbinden der Datei *FILE2.H* dann vom vorherigen Einbinden der Datei *FILE1.H* abhängig. Und solche Abhängigkeiten führen früher oder später zu einer nicht mehr überschaubaren Abhängigkeit. Aber keine Panik, denn jetzt kommt die Lösung dieses Problems, mithilfe der beiden Anweisungen #*ifndef* und #*define*. Schließen Sie in Zukunft jede einbindbare Datei in einen #*ifndef...#endif* Zweig ein, so wie im Beispiel angegeben. Als abzufragenden Symbolnamen sollten Sie irgendwie den Namen der entsprechenden einzubindenden Datei mit verwenden. Wurde die Datei noch nicht eingebunden, d.h. das Symbol ist noch nicht definiert, so definieren Sie das Symbol mit der #*define*-Anweisung und führen anschließend wie gewohnt die Definitionen und Deklarationen durch. Wird die Datei dann ein zweites Mal eingebunden, so ist das entsprechende Symbol bereits definiert, und damit wird der Inhalt der Datei einfach übersprungen.

Beispiel 7. #ifndef

```
Datei FILE1.H
#ifndef FILE1_H
#define FILE1_H
.... // Definition und Deklaration
#endif
Datei FILE2.H
#ifndef FILE2_H
#define FILE2_H
#include "FILE1.H"
.... // weitere Definitionen und Deklarationen
#endif
Quellcode-Datei MAIN.CPP
#include "FILE1.H"
#include "FILE2.H"
....
```

9.4. #if, #elif, #else und #endif

Außer der einfache Abfrage ob ein Symbol definiert ist oder nicht, gibt es auch ein IF-ELIF-ELSE-ENDIF Konstrukt, das im Prinzip genauso arbeitet wie die verwandte C++ Verzweigung. Der Unterschied zwischen der Präprozessor-Verzweigung und der C++ Anweisung ist der, dass die Präprozessor-Verzweigung zum einen vor dem Compilerdurchlauf ausgewertet wird und zum anderen auch nur mit Präprozessor-Symbolen arbeitet, also nicht C++ Daten. Im Beispiel rechts werden je nach Wert des Symbols *MAX* verschiedene Text ausgegeben. Neu ist die #*elif* Anweisung, sie ist eine Mischung aus der Anweisungsfolge ELSE und IF.

Die *#if* und *#elif* Anweisungen können auch mehrere Ausdrücke als Bedingungen auswerten. Die einzelnen Bedingungen werden dann entweder durch den Operator || verodert oder durch && verundet.

Beispiel 8. #if, #elif, #else und #endif

```
#define MAX 10
char array[MAX];

int main()
{
    #if MAX<10
        cout << "kleines Feld";
    #elif MAX == 10
        cout << "10er-Feld";
    #elif MAX < 50
        cout << "mittleres Feld";
    #else
        cout << "grosses Feld";
    #endif
    cout << endl;
}
```

9.5.defined

Ebenfalls im Zusammenhang mit der IF-Präprozessor-Anweisung steht die Anweisung *defined(...)*. *defined(...)* dient zum Überprüfen ob ein Symbol definiert ist. Das zu überprüfende Symbol wird dann innerhalb einer Klammer angegeben. *defined(...)* liefert *1* zurück, wenn das Symbol definiert ist und ansonsten *0*. Vor *defined(...)* kann noch der Operator '!' stehen um das Abfrageergebnis zu negieren. Im Beispiel werden je nach verwendetem Compiler verschiedene Ausgaben mit ins Programm übernommen. Die in den *defined(...)* Anweisungen stehenden Symbole (beginnend mit 2 Underscore und einem nachfolgenden Großbuchstaben!) sind Symbole die vom Compiler selbst definiert werden. So definiert der BORLAND-Compiler z.B. das Symbol *_BORLANDC_* während der MICROSOFT Compiler das Symbol *_MSVC_* definiert. Welche Symbole Ihr Compiler definiert, entnehmen Sie bitte aus der Dokumentation zum Compiler.

Beispiel 9. defined

```
int main ()
{
#if defined(_BORLANDC_)
    cout << "Mit Borland übersetzt!";
#elif definied (_MSVC_)
    cout << "Mit VC++ übersetzt!";
#else
    cout << "Den kenn ich nicht!";
#endif
   ....
}
```

Auch die Sprache C++ selbst definiert einige Symbole die Sie der folgenden Tabelle entnehmen können:

Tabelle 1. In C++ definierten Symbole

Symbol	Bedeutung
__LINE__	Enthält aktuelle Zeilennummer im Quellcode
__FILE__	String mit dem Namen der akt. Datei
__DATE__	String mit dem aktuellen Datum in der Form Monat/Tag/Jahr
__TIME__	String mit der aktuellen Uhrzeit in der Form Stunde:Minute:Sekunde
__STDC__	Compilerspezifisch, in der Regel ist dieses Symbol definiert wenn nur ANSI C/C++ Code vom Compiler akzeptiert wird.
__cplusplus	ANSI C++ konforme Compiler definieren für dieses Symbol einen Wert mit mind. 6 Ziffern, alle anderen C++ Compiler einen Wert mit bis zu 5 Ziffern. C Compiler definieren dieses Symbol überhaupt nicht.