Kapitel 15

Präprozessor Anweisungen, Makros, Namensbereiche

Präprozessor Anweisungen

- Die Präprozessor Anweisungen werden immer mit dem Doppelgatter # eingeleitet.
- Der Präprozessor erzeugt – entsprechend der Anweisung – den Quelltext, der dann vom Compiler übersetzt wird.
- Am Ende einer Präprozessor Anweisung steht **kein** Semikolon.
- Präprozessor Anweisungen sind Anweisungen, die vor dem eigentlichen Übersetzen des Programms durch den Präprozessor ausgeführt werden.
- Alle Zeilen, die mit dem Doppelgatter beginnen, werden als Präprozessor Anweisung interpretiert.

include Anweisung

- Der Quelltext von Header-Dateien kann mit der *include* Anweisung eingebunden werden.
- Header-Dateien sind Textdateien, die u.a. Deklarationen enthalten.
- Es gibt zwei Schreibweisen:

```
#include <Dateiname>
#include "Dateiname.h"
```

- Spitze Klammern werden bei Standard-Header-Dateien verwendet (zB. iostream, cmath).

include Anweisung

- Die Dateien werden in einem von der Entwicklungsumgebung voreingestellten Pfad gesucht.
- Zur Einbindung von selbst verfassten Header-Dateien verwendet man Anführungszeichen.
- Sie werden im aktuellen Arbeitsverzeichnis gesucht.
- Es können absolute und relative Pfade angegeben werden:

```
#include "D:\Entwicklung\CPP\include\const.h"
#include "..\include\const.h"
```

include Anweisung

- Alle in einer Header-Datei deklarierten Namen sind global verfügbar.
- Das kann bei großen Programmen zu Namenskonflikten führen.
- Daher gibt es in C++ z.B. zur Header-Datei *math.h* noch eine Header-Datei *cmath*, die dieselben Namen in einem Namensbereich *std* deklariert.

include Anweisung

```
#include <math.h>
```

- ist äquivalent zu:

```
#include <cmath>
using namespace std;
```

define Anweisung

- Mit der *define* Anweisung werden symbolische Konstanten und Makros definiert.
- Es werden drei Formen unterschieden:
  - Definition eines Symbols
  - Definition eines Symbols mit Wertzuweisung
  - Definition eines Makros

Definition eines Symbols

- Das Symbol (z.B. DEBUG) ist ein beliebiger Name.
- Diese Definition kann in anderen Präprozessor Anweisungen (z.B. *#ifdef*) abgefragt werden.

```
#define DEBUG
```

Definition eines Symbols mit Wertzuweisung

- Es wird ein Symbol definiert und diesem ein Wert zugewiesen.

```
#define MAXINDEX  10
```

- Der Präprozessor ersetzt das Wort MAXINDEX vor dem Compilerlauf durch 10.

Definition eines Symbols mit Wertzuweisung

- Von der Ersetzung ausgenommen sind Strings und Kommentare.
- Zur Definition von Konstanten ist es in jedem Fall besser, benannte Konstanten zu verwenden, anstatt sie mit *define* zu definieren

```
const int MAXINDEX = 10;
```

Definition eines Makros

- Ein Makro ist eine Abkürzung für eine beliebige Anweisung.

```
#define BEEP  cout << "\a"
```

- Trifft der Präprozessor auf das Wort Symbol (hier: BEEP), wird dieses durch die bei der Makrodefinition angegebenen Anweisungen ersetzt.

Definition eines Makros

- Erstrecken sich die Anweisungen eines Makros über mehrere Zeilen, sind die Zeilen mit einem Backslash abzuschließen.
- Makros können auch Parameter besitzen.
- Die Parameter werden hier - genauso, wie bei Funktionen - in Klammern eingeschlossen und durch Kommata getrennt. Datentypen werden keine angegeben.

Definition eines Makros

```
#include <iostream>     using namespace std;
#define BEEP cout "\a"
#define MOD(a,b)   ((a)%(b))
#define QUAD(a)    ((a) * (a))
void main()
{
      BEEP;
      cout << MOD(123, 11) << endl;
      cout << QUAD(17) << endl;
}
```

```
Ton!
2
289
```

undef Anweisung

- Mit der *undef* Anweisung kann ein definiertes Symbol wieder gelöscht werden.

```
#undef DEBUG
```

ifdef, ifndef und *endif* Anweisung

- Mit der Präprozessor Anweisung *ifdef* kann abgefragt werden, ob ein bestimmtes Symbol definiert ist oder nicht.
- Ist das Symbol definiert, übernimmt der Präprozessor alle Anweisungen die zwischen der *ifdef* Anweisung und der dazugehörigen *endif* Anweisung stehen.

ifdef, ifndef und *endif* Anweisung

```
#define DEBUG
#ifdef DEBUG
    cout << "Debug Version" << endl;
#endif
#undef DEBUG
#ifndef DEBUG
    cout << "Release Version" << endl;
#endif
```

```
Debug Version
```

```
Release Version
```

if, elif, else und *endif* Anweisung

- Diese Anweisungen bieten erweiterte Möglichkeiten.
- Sie arbeiten ähnlich, wie die Verzweigung in C++.

```
#define INTBITS  32
#if INTBITS == 16
    cout << "16 Bit int" << endl;
#elif INTBITS == 32
    cout << "32 Bit int" << endl;
#else
    cout << "INTBITS undef oder falsch!" << endl;
#endif
```

```
32 Bit int
```

if, elif, else und *endif* Anweisung

- Die *if* und *elif* Anweisungen können auch mehrere Ausdrücke als Bedingungen auswerten.
- Die einzelnen Bedingungen können dann mit dem Oder-Operator || oder dem UND-Operator && verknüpft werden.

defined Operator

- Mit dem *defined* Operator kann überprüft werden, ob eine symbolische Konstante oder ein Makro definiert ist.
- Vor *defined* kann noch der Operator ! stehen, um das Ergebnis der Abfrage zu negieren.

```
#if !defined(_WIN32) && !defined(_MAC)      // aus math.h
    #error ERROR: Only Mac or Win32 targets supported!
#endif
```

error Anweisung

- Diese Anweisung bewirkt einen sofortigen Abbruch des Compilervorgangs.
- Nach der Präprozessor Anweisung kann noch ein Text stehen.
- Der Text muss nicht in Anführungszeichen stehen und wird ausgegeben.

```
#if  !defined(__cplusplus)
    #error C++ compiler required.
#endif
```

pragma Anweisung

- Mit der *pragma* Anweisung wird der Compiler angewiesen, eine (implementationsspezifische) Operation auszuführen.
- Nicht alle Operationen, die nach dem Schlüsselwort *pragma* folgen, sind bei allen Compilern bekannt.
- Eine, dem Compiler nicht bekannte *pragma* Operation, wird vom Präprozessor einfach ignoriert.

```
#if  _M_IX86 == 500
    #pragma message("Erstellung für Pentium-Prozessor")
#endif
```

pragma Anweisung

- Mit *pragma message* kann bei Microsoft Visual C++ eine Meldung ausgegeben werden, ohne dass der Compilervorgang abgebrochen wird.
- Der Text muss eingeklammert und in Anführungszeichen stehen.
- Außer der Operation *message* gibt es noch ein paar weitere *pragma* Operationen. Diese können sich aber von Compiler zu Compiler unterscheiden.

Namensbereiche

- Bei großen Programmen kommt es bei der Vergabe von globalen Variablen häufig zu Konflikten, wenn gleiche Namen verwendet werden.
- Namensbereiche werden zur Vermeidung solcher Konflikte eingesetzt.
- Innerhalb eines Namensbereiches können Namen verwendet werden, auch wenn sie außerhalb bereits definiert wurden.
- Der globale Geltungsbereich wird somit in Teilbereiche unterteilt.

Namensbereiche

- Wird von außerhalb auf die Elemente eines Namensbereichs zugegriffen, muss zusätzlich der Namensbereich angegeben werden.
- Dieser wird dem Elementnamen mit dem Bereichsoperator :: vorangestellt.

```
#include <iostream>
void main()
{
    std::cout << "Text" << std::endl;
}
```

Namensbereiche

- Mit der *using* Anweisung kann explizit der Namensbereich einer Variablen oder Funktion für die restlichen Anweisungen des aktuellen Blocks vorgegeben werden.

```
#include <iostream>
void main()
{
    using std::cout;
    using std::endl;
    cout << "Text" << endl;
}
```

Namensbereiche

- Sollen aber alle Elemente eines Namensbereichs eingebunden werden, wird die bereits häufig gezeigte Form der *using* Anweisung verwendet.

```
#include <iostream>
using namespace std;
void main()
{
    cout << "Text" << endl;
}
```