57. Namensräume

Namensräume wurden geschaffen um Namenskonflikte von Variablen, Klassen usw. aufzulösen. Auch diese Problematik lässt sich wiederum am besten anhand eines Beispiels demonstrieren.

Im Beispiel wird eine Variable *var* und eine Klasse *CAny* definiert. Auf beide wird in *main()* zugegriffen. Bis jetzt also noch nichts Ungewöhnliches.

```
1:  // Definitionen
2:  int var;
3:  class CAny
4:  {...};
5:  // main() Funktion
6:  int main()
7:  {
8:      CAny myObj;
9:      auto var = 10;
10:     ...
11: }
```

Nehmen wir jetzt einmal an, wir erweitern das Programm und setzen dazu eine Klassenbibliothek ein. Zu dieser Klassenbibliothek gehört eine Header-Datei (im Beispiel *clib.h*), die auszugsweise unten dargestellt ist. In dieser Header-Datei sind unglücklicherweise ebenfalls eine Klasse *CAny* und eine Variable *var* definiert.

```
1:  // Headerdatei der Klassenbibliothek clib
2:  #ifndef CLIB_H
3:  #define CLIB_H
4:  short var;
5:  class CAny
6:  {...};
7:  #endif
```

Was folgt, lässt sich schon erahnen. Der Compiler bzw. der Linker wird sich beim Übersetzen des Programms beschweren, dass einige Symbole mehrfach definiert sind.

```
1:  #include "clib.h"
2:  // Definitionen
3:  int var; // Das erzeugt jetzt Fehler!
4:  class CAny
5:  {...};
6:  // main() Funktion
7:  int main()
8:  {
9:      CAny myObj;
10:     auto var = 10;
11:     ...
12: }
```

57.1 Definition eines Namensraums

Und für solche Fälle wurde der Namensraum geschaffen. Die Syntax zur Definition eines Namensraums lautet:

```
namespace [NName]
{
    ... // Definitionen und Deklarationen
}
```

NName definiert den Namen des Namensraums. Und alle in einem Namensraum stehenden Definitionen und Deklarationen sind nur innerhalb dieses Namensraums gültig! Gleichnamige Definitionen und Deklarationen in unterschiedlichen Namensräumen erzeugen somit keinen Namenskonflikt mehr.

Angemerkt werden soll an dieser Stelle noch, dass für einen Namensraum Attribute angegeben werden können. So markierte die Anweisung

```
namespace [[deprecated]] myspace
{
    ...
}
```

den Namensraum *myspace* als veraltet. Die Anwendung wird sich zwar erstellen lassen, jedoch wird beim Übersetzen eine Meldung ausgegeben, dass der Namensraum veraltet ist. Eine Beschreibung der Attribute ist unter 31.9 Attribute zu finden.

57.2 Anwendung von Namensräumen

Um die Namenskonflikte in unserem Beispiel mithilfe von Namensräumen aufzulösen, stehen zwei Lösungswege zur Verfügung. Im ersten Fall wird die Header-Datei *clib.h* in einen eigenen Namensraum gelegt und im zweiten Fall die Definitionen und Deklarationen der Anwendung. Beide Lösungen sind von der Wirkungsweise gleichwertig. Da die Definitionen von *var* und *CAny* nun in verschiedenen Namensräumen liegen, kommt es zu keinen Namenskonflikten mehr.

Lösung 1:

```
1:  namespace clib
2:  {
3:      #include "clib.h"
4:  }
5:  int var;
6:  class CAny
7:  {...};
```

Lösung 2:

```
1:  #include "clib.h"
2:  namespace cppkurs
3:  {
4:      int var;
5:      class CAny
6:      {...};
7:  }
8:  // Hier geht's mit main() weiter
```

Wird ein bereits bestehender Namensraum erneut definiert, wird der bisherige Namensraum erweitert. Im Beispiel enthält der Namensraum *cppkurs* alle Deklarationen und Definitionen aus den beiden Header-Dateien *mylib.h* und *yourlib.h*.

Header-Datei mylib.h

```
1:  namespace cppkurs
2:  {
3:     ...
4:  }
```

Header-Datei yourlib.h

```
1:  namespace cppkurs
2:  {
3:     ...
4:  }
```

57.3 Zugriff auf Namensräume

Sehen wir uns an, wie auf Definitionen und Deklarationen in Namensräumen zugegriffen wird. Hierzu gibt es verschiedene Möglichkeiten. Eine Möglichkeit ist, beim Zugriff zuerst den jeweiligen Namensraum, dann den Zugriffsoperator `::` und danach die Deklaration bzw. Definition anzugeben.

```
1:  namespace clib
2:  {
3:      #include "clib.h"
4:  }
5:  int var; // Eigene Definitionen
6:  class CAny
7:  {...};
8:  // Zugriff auf var aus Namensraum clib
9:  clib::var = 10;
10: // CAny aus dem Namensraum clib
11: clib::CAny myObj;
12: // Zugriff auf var im globalen Namensraum
13: var = 33;
```

Eine weitere Möglichkeit ist, den Namensraum einer Variablen, Funktion, Objekt usw. einzublenden. Dies erfolgt mittels des Schlüsselworts *using*.

```
using NName::DName;
```

NName ist der Name des Namensraums und *DName* der Name der jeweiligen Deklaration bzw. Definition. Die *using*-Anweisung erlaubt nur die Angabe eines Namens. Sollen mehrere Namen aus einem Namensraum eingeblendet werden, sind mehrere *using*-Anweisungen hierfür zu verwenden.

```
1:  // Für CAny standardmäßig clib-Namensraum setzen
2:  using clib::CAny;
3:  // Objekt aus clib-Namensraum definieren
4:  CAny myObj;
5:  // Objekt aus globalem Namensraum definieren
6:  ::CAny anotherObj;
7:  // Standard var ist aus clib Namensraum
8:  using clib::var;
9:  var = 10;
10: // nun var aus globalem Namensraum setzen
11: ::var = -10;
```

Und die dritte Möglichkeit ist, alle Namen eines Namensraums einzublenden. Dazu wird folgende *using*-Direktive verwendet:

```
using namespace NName;
```

NName ist wiederum der Name des einzublendenden Namensraums.

In einem Programm können durchaus mehrere Namensräume mittels *using namespace* eingeblendet werden. Diese Namensräume sind dann additiv, d.h., es sind alle Member aus den Namensräumen gültig.

57.4 Geschachtelte Namensräume

Namensräume können auch geschachtelt werden. Beim Zugriff auf einen Namen müssen dann beide Namensräume (von außen nach innen) angegeben werden.

```
1:  namespace outer
2:  {
3:      namespace inner
4:      {
5:          int var;
6:      }
7:  }
8:  // Expliziter Zugriff auf var
9:  outer::inner::var = 10;
```

Ab C++17 kann ein geschachtelter Namensraum auch außerhalb des umschließenden Namensraums definiert werden. Dazu wird bei der Definition des inneren Namensraums der äußere Namensraum mit angegeben.

```
1:  namespace outer
2:  {
3:  ...
4:  }
5:  // Innerer Namensraum vom outer
6:  namespace outer::inner
7:  {
8:     int var;
9:  }
10:
11: // Expliziter Zugriff auf var
12: outer::inner::var = 10;
```

57.5 Anonyme Namensräume

Mithilfe eines anonymen Namensraums können Variablen, Funktionen usw. einer Übersetzungseinheit (in der Regel ist dies eine Quelldatei) nach außen hin verborgen werden. Innerhalb der Übersetzungseinheit kann weiterhin auf alle Daten und Funktionen des Namensraums wie gewohnt zugegriffen werden, aber außerhalb der Übersetzungseinheit besteht kein Zugriff darauf. So verhält sich im Beispiel die Variable *var* innerhalb der Quelldatei wie eine globale Variable, ist aber außerhalb der Übersetzungseinheit nicht sichtbar. Das gleiche Verhalten kann alternativ auch durch die Definition der Variable *var* als *static* Variable erreicht werden.

```
1:  namespace
2:  {
3:      int var;
4:      ...
5:      void Func1(...)
6:      {
7:          var = 10;
8:          ...
9:      }
10:     ...
11: }
```

Der Unterschied zwischen einem anonymen Namensraum und *static* ist, dass in einem anonymen Namensraum auch Datentypen deklariert werden können.

57.6 Inline-Namensraum

Ein *inline*-Namensraum ist ein Namensraum, der innerhalb eines anderen Namensraums definiert ist und dadurch gekennzeichnet ist, dass vor dem Schlüsselwort *namespace* das Schlüsselwort *inline* steht. Damit ergibt sich folgender prinzipieller Aufbau:

```
1:  namespace myNamespace
2:  {
3:      inline namespace newVersion
4:      {
5:          ...
6:      }
7:  }
```

Auf alle Namen in einem *inline*-Namensraum kann ohne Angabe des Namens des *inline*-Namensraums zugegriffen werden.

Wofür ein solcher *inline*-Namensraum sinnvoll eingesetzt werden kann sehen wir uns an einem Beispiel wieder an. Nehmen wir an, im Namensraum *myLibs* sei eine Funktion *PrintValue()* definiert.

```
1:  // Umgebender Namensraum
2:  namespace myLibs
3:  {
4:      void PrintValue(int x)
5:      {
6:          cout << "Std-Print():" << x << endl;
7:      }
8:  }
9:
10: int main()
11: {
12:     MyLibs::PrintValue(5);
13: }
```

Diese Funktion soll durch eine neuere Version ersetzt werden. Dabei soll jedoch die ursprüngliche Funktion aus Kompatibilitätsgründen weiterhin erhalten bleiben und vom Anwender bei Bedarf explizit aufgerufen werden können. Das Problem, das sich hier ergibt, ist, dass die alte und die neue Funktion die gleiche Signatur (Funktionsname und -parameter) haben und sie deswegen nicht im gleichen Namensraum liegen dürfen.

Um dieses Problem zu lösen, werden zunächst zwei neue Namensräume eingefügt. Ein Namensraum enthält die alte Funktion und der andere Namensraum die neue Funktion. Damit standardmäßig die neue Funktion aufgerufen wird, wird der Namensraum der neuen Funktion als *inline*-Namensraum definiert. Und wie Sie wissen, kann dann der Name des *inline*-Namensraums beim Aufruf der neuen Funktion weggelassen werden.

```
1:  // Umgebender Namensraum
2:  namespace myLibs
3:  {
4:      // Version 1 der PrintValue Funktion
5:      namespace Version1
6:      {
7:          void PrintValue(int x)
8:          {
9:              cout << "Std-Print():" << x << endl;
10:         }
11:     }
12:     // inline Namensraum: neue Version 2 der Funktion
13:     inline namespace Version2
14:     {
15:         void PrintValue(int x)
16:         {
17:             cout << "New-Print():" << x << endl;
18:         }
19:     }
20: }
21:
22: int main()
23: {
24:     // Funktion Version2 aufrufen
25:     MyLibs::PrintValue(5);
26:     // Funktion Version1 aufrufen
27:     MyLibs::Version1::PrintValue(10);
28: }
```

Soll standardmäßig die alte Funktion aufgerufen werden, ist lediglich der Namensraum *Version1* anstelle des Namensraums *Version2* als *inline*-Namensraum zu definieren.