C++

- Funktionen und mehr -

Friedrich-Schiller-Universität Jena

Kerstin Gößner und Ralf Wondratschek

Prof. Dr. habil. Wolfram Amme
Dipl.-Inf. Thomas Heinze

Inhaltsverzeichnis

1 Einleitung

Eines der wichtigsten Konzepte in der Softwareentwicklung ist das Modularisieren des Programmcodes. Anstatt mehrere tausend Zeilen von Code hintereinander zu schreiben, wird dieser in einzelne Teile zerlegt und das Programm setzt sich aus diesen Stücken zusammen. Die Vorteile liegen klar auf der Hand: Wiederverwendbarkeit, Übersichtlichkeit, bessere Wartbarkeit, Fehlervermeidung und viele mehr.
Dieses Prinzip findet sich in fast allen Programmiersprachen wieder, sei es Assembler, Pascal, Java und C. In diesem Bericht soll die Umsetzung für die Programmiersprache C++ genauer beleuchtet werden, angefangen bei allgemein gültigen Konzepten wie Deklarieren, Namensbereiche und Funktionen bis hin zu C++ spezifischen Konstrukten wie Headerdateien und Using-Deklarationen.

2 Deklaration, Definition und Initialisierung

2.1 Deklaration

Bei einer Deklaration handelt es sich um eine Anweisung, die einem Namen einen Gültigkeitsbereich einführt und einen Typ für das benannte Element und optional einen Initialisierer angibt.
Man unterscheidet zwar zwischen einer Deklaration einer Funktion und Variablen, dennoch sind die Bedingungen gleich. Formal sieht eine Deklaration wie folgt aus:
Variable:

<extern / const> [Typ] [Name] <Initialisierer> ;

Funktion:

[Rückgabetyp] [Name] (<Liste formaler Argumente>);

Der Name ist frei wählbar. Über ihn wird eine Variable bzw. Funktion angesprochen. In einem Gültigkeitsbereich muss eine Variable eindeutig sein, d.h., dass keine zwei Variablen mit gleichem Namen deklariert werden dürfen. Etwas Ähnliches gilt für Funktionen, dies wird später allerdings genauer erklärt.
Beim Typ bzw. Rückgabetyp handelt es sich um eine dem Compiler bekannte Datenstruktur.
Durch eine Deklaration weiß der Compiler, dass eine Variable bzw. Funktion existiert. Am häufigsten sind Deklarationen in Headerdateien zu finden, was in dem entsprechenden Kapitel später genau erläutert wird.
Wird eine normale Variablendeklaration um das Schlüsselwort “const“ erweitert, handelt es sich um eine Konstantendeklaration. Der Variablen darf zur Laufzeit nur ein einziger Wert zugewiesen werden.
Bespiele dafür zeigt Abbildung 1

2.2 Definition

Eine Definition ist eine Deklaration, die das Element vollständig spezifiziert. Sie gibt an, worauf sich ein Name konkret bezieht. Im Gegensatz zu einer reinen Deklaration wird für eine Definition Speicher reserviert. Somit erklärt sich auch, warum es in einem Gültigkeitsbereich keine doppelten Definitionen, allerdings exakt gleiche Deklarationen geben darf.
Eine Definition wird wie folgt angegeben:
Variable:

[Typ] [Name] <Initialisierer>;

Funktion:

[Rückgabetyp] [Name] (<Liste formaler Argumente>) <Anweisungen>

2.3 Initialisierung

Bei einer Initialisierung handelt es sich implizit auch um eine Definition. Durch eine Initialisierung wird eine Variable mit einem bestimmten Wert vorbelegt.

Eine Ausnahme stellt dabei folgendes Konstrukt dar:

extern [Typ] [Name] = [Wert];

Ein Beispiel: Abbildung 2

Es handelt sich korrekterweise um eine Initialisierung, dennoch muss die Variable an anderer Stelle definiert wurden sein, da durch das Schlüsselwort extern kein Speicher zugewiesen wird.

Für einen gut lesbaren Code sollten Variablen stets bei ihrer Definition auch initialisiert werden. So vermeidet man zusätzlich die Gefahr von Spaghetti-Code.

Vorsicht sei auch bei der Standardinitialisierung geboten. Eingebaute Typen wie “int“ und “double“ werden global automatisch mit 0 initialisiert, lokal und als Klassenmember jedoch nicht. Eine Standardinitialisierung erfolgt über den Standardkonstruktor.

Eine Zusammenfassung befindet sich im Anhang (Abbildung 3).

3 Funktionsaufrufe und -rückgabewerte

3.1 Rückgabewerte

Wurde eine Funktion mit einem Rückgabetyp deklariert, braucht der Funktionsrumpf der Definition stets eine “return“ Anweisung. Der Typ der zurück gegebenen Variablen muss dem Rückgabetyp entsprechen. Auch bei tiefen Verschachtelungen in der Funktion darf die Rückgabe nicht vergessen werden.
Eine Funktion, bei der keine Rückgabe erforderlich, wird statt mit einem Rückgabetyp dem Schlüsselwort “void“ deklariert. In anderen Programmiersprachen entspricht eine solche Funktion einer Prozedur, jedoch wird in C++ darin nicht unterschieden.
Ein Beispiel befindet sich im Anhang, in Abbildung 4.

3.2 Liste formaler Argumente

Die Liste formaler Argumente befindet sich sowohl bei der Deklaration als auch bei der Definition einer Funktion zwischen zwei runden Klammern. Häufig bezeichnet man ein Argument der Liste auch als Parameter.
In dem Gültigkeitsbereich der Funktion entsprechen die Parameter lokalen Variablen. Somit kann man das Übergeben eines Arguments implizit gleichsetzen mit dem Initialisieren des Parameters mit diesem Argument.

3.3 Funktionszuordnung

Wird eine Funktion aufgerufen, muss der Funktionsrumpf aus der Definition im Hauptspeicher gefunden werden. Die Zuordnung findet anhand einer Signatur der Funktion statt. Diese Signatur setzt sich aus dem Namen und der Liste der formalen Argumente zusammen, jedoch nicht dem Rückgabetyp.
Dieser Zusammenhang erlaubt das Überladen von Funktionen. Als überladen werden Funktionen bezeichnet, die den gleichen Rückgabetyp und Namen jedoch eine unterschiedliche Liste von Parametern haben.
Ein Beispiel hierzu ist in Abbildung 5 zu sehen.
Das Weglassen des Rückgabetyps in der Signatur ergibt wahrscheinlich erst beim zweiten Blick einen Sinn. Betrachtet man Abbildung 6, so kann man nicht eindeutig entscheiden, welche Funktion aufgerufen wird! Der Rückgabetyp nimmt also keinen Einfluss.

3.4 Parameterübergabe

In C++ gibt es drei typische Methoden Parameter zu übergeben: Pass-by-value, Pass-by-reference und Pass-by-const-reference.
Bei Pass-by-value wird nur der Wert des Arguments übergeben, jedoch nicht das Argument selbst. Da der Parameter in der Funktion einer lokalen Variablen entspricht und

diese beim Funktionsaufruf initialisiert wird, kann man das Übergeben als Kopieren des Arguments verstehen. Pass-by-value bietet sich somit für kleine Werte an, da es einfach zu nutzen ist, allerdings sollte es bei großen Argumenten vermieden werden, da für die Parameter neuer Speicher und Rechenzeit beim Kopieren verbraucht wird.(Beispiel: Abbildung 7)

Wie der Name schon sagt, wird bei Pass-by-reference eine Referenz statt einer Kopie übergeben. Der Parameter wirkt somit als Alias und zeigt auf das Argument. Signalisiert wird dies durch ein kaufmännisches Und. Ändert man diese Referenz, ändert sich auch das Argument, dass man übergeben hat. Somit sollte man stets bei der Übergabe eines Arguments mit Pass-by-reference davon ausgehen, dass das Objekt verändert wird!(Beispiel: Abbildung 8)

Möchte man dennoch eine Referenz übergeben, aber signalisieren, dass diese Referenz nicht verändert wird, verwendet man Pass-by-const-reference. Vor dem Referenzparameter steht somit noch das Schlüsselwort "const". Dieses Vorgehen ist sehr effizient, weil Rechenzeit zum Kopieren und Speicher für die Kopie gespart wird. Zusätzlich kann man davon ausgehen, dass nach dem Funktionsaufruf das übergebene Objekt unverändert ist. (Beispiel: autorefbsp9)

3.5 Implementierung von Funktionsaufrufen

Bei einem Funktionsaufruf wird ein "Function activation record" angelegt. Dabei handelt es sich um eine Datenstruktur, die die lokalen Variablen (somit auch die Parameter) und Implementierungsdetails, z.B. eine Rücksprungadresse, beinhaltet. Die Kosten für Anlegung sind nicht von der Größe abhängig. Beispielsweise werden bei Verschachtelungen unter Umständen lokale Variablen nicht angelegt.
Jeder Datensatz ist voneinander unabhängig und abgeschlossen. Dies ist wichtig mehrfachen Aufrufen einer Funktion, wie es bei der Rekursion der Fall ist. Ein Datensatz wird auf dem Stack gespeichert, welcher nach dem Last In First Out Prinzip wächst und schrumpft.

4 Headerdateien und deren Verwendung

4.1 Headerdateien

Ein Header ist eine Datei mit einer Zusammenfassung mehrerer Deklarationen und anderen Quellcodes, welche in mehreren Programmen verwendet werden sollen.
Der Name einer Headerdatei ist folgendermaßen aufgebaut:

[filename].h

Hat man eine Headerdatei erstellt oder eine für die eigene Implementierung nützliche Headerdatei gefunden, muss man sie lediglich in den eigenen Quellcode, also in eine Datei

mit .cpp Endung einbinden. Dies wird mit Hilfe von #include (siehe Unterabschnitt 4.2) ermöglicht.

4.2 #include-Anweisung

Mit der #include-Anweisung ist es möglich, den Code aus einer Headerdatei [filename1.h] in eine C++ - Datei [filename2].cpp einzubinden. Dabei entspricht die Anweisung #include [fileName1.h] in der C++ - Datei [filename2.cpp] dem Kopieren der Textzeilen aus filename1.h in die Datei filename2.h.
Zeilen mit #include werden immer zuerst bearbeitet, da der Präprozessor, ein Teil des Compilers zuerst den Quellcode durchläuft und dabei nur die Zeilen mit einem # am Beginn übersetzt (sog. Präprozessor-Direktiven). Erst danach wird auch der restliche Quelltext übersetzt. Es darf nur eine #include-Anweisung pro Zeile vorhanden sein.
Weiterhin ist zu beachten, dass #include Anweisungen in allen Dateien, die Teile des Headers verwenden, eingefügt werden müssen.
In Abbildung 10 wird die Verwendung und der Nutzen von Headern dargestellt. Die Bibliothek “std_lib_faclities“ wurde im Buch ... als Standardbibliothek verwendet.

5 Scopes - Gültigkeitsbereiche

5.1 Definition

Ein Scope ist ein bestimmter Abschnitt im Programmcode. Wenn eine Variable in einem Scope deklariert wurde, ist sie auch nur bis zum Ende des Scopes gültig. Alle Variablen in einem bestimmten Scope sind auch in allen eingeschlossenen Bereichen gültig. Scopes kann man sowohl explizit also auch implizit definieren.

5.2 Arten von Scopes

- Globaler Scope
  Dieser Gültigkeitsbereich liegt über allen, er kann also nicht eingeschlossen werden und ist implizit definiert.

- Namensbereich
  Ein Namensbereich ist ein explizit benannter Scope, der entweder innerhalb des globalen Scopes oder in einem anderen Namensbereich liegt. Genauere Erklärungen erfolgen in Abschnitt 7.

- Klassenbereich
  In einer Klasse deklarierte Variablen sind auch nur in dieser Klasse gültig. (Beispiel:

Abbildung 11)

- Lokaler Gütligkeitsbereich
  Wird eine Variable innerhalb von -Klammern von Funktionen oder Anweisungen deklariert, ist sie auch nur dort gültig. (Beispiel: Abbildung 12)

- Anweisungsbereich
  Wird eine Variable im Kopf einer for-Schleife oder einer if-Anweisung deklariert, dann ist diese Variable nur innerhalb der for-Schleife bzw if-Anweisung gültig. (Beispiel: Abbildung 13)

5.3 Zweck

Wie schon erwähnt, ist die Aufgabe von Scopes vor allem die Ermöglichung einer möglichst großen Lokalität von Variablen. So ist es möglich, Namen für Variablen wieder zu verwenden.
Außerdem wird durch die Beschränkung der Gültigkeit die Möglichkeit von Kollisionen vermieden.

6 Verschachtelung

Man kann C++-Konstrukte, die Scopes definieren, ineinander verschachteln.
Es folgt eine kurze Erläuterung, welche Konstruktionen überhaupt möglich sind, und welche benutzt oder möglichst vermieden werden sollten.

- Funktionen innerhalb von Klassen
  Dies ist der häufigste Fall einer Verschachtelung. Es gibt eine Klasse in der eine oder mehrere Funktionen deklariert sind. Die Funktionen werden dann über Instanzen der Klasse aufgerufen und erfüllen eine bestimmte Aufgabe. Dabei kann man die Funktionen entweder innerhalb der Klasse definieren, oder auch nur innerhalb der Klasse deklarieren und außerhalb definieren. Beide Möglichkeiten sind in Abbildung 14 dargestellt.

- Memberklassen
  Memberklassen sind Klassen, die sich innerhalb von anderen Klassen befinden. Dabei kann man eine Hierarchie mit beliebig vielen Stufen bilden. Im Beispiel Abbildung 15 ist eine Verschachtelung mit 3 Klassen dargestellt.

  Dieses Konstrukt ist meist für die Implementierung komplizierter Klassen hilfreich.

- Lokale Klassen
  Lokale Klassen werden innerhalb von Funktionen definiert. Das bedeutet, dass sie auch nur innerhalb dieser Funktion gültig sind. Dabei können auch mehrere lokale Klassen in einer Funktion definiert werden.
  Diese Konstruktion ist sehr unübersichtlich und sollte deshalb möglichst vermieden werden. Die eigentlich Funktion wird viel zu lang und sollte besser in mehrere kleinere Funktionen aufgesplittet werden. (Beispiel: Abbildung 16)

- Eingebettete Funktionen
  Hier ist eine Funktion in einer anderen Funktion definiert.
  Diese Konstruktion ist in C++ nicht erlaubt und wird durch den Compiler zurück gewiesen (siehe auch Abbildung 17).

- Verschachtelte Blöcke
  Diese Konstruktion ist nicht zu vermeiden, da es in den meisten Funktionen unmöglich ist, das gewünschte Ergebnis zu erzielen, ohne mehrere Schleifen und Anweisungen ineinander zu schachteln.

7 Namespaces - Namensbereiche

Eine Namespace ist eine Gruppierung von Deklarationen. So kann man Klassen, Funktionen, Daten und Typen, die zusammen gehören zu einem benannten Teil des Programmes zusamen fassen.

Beim Verwenden des globalen Gültigkeitsbereiches kommt es schnell zu Kollisionen, vor allem dann, wenn mehrere Programmierer an einem Projekt beteiligt sind. Bei Verwendung von namespaces lassen sich solche Gefahren umgehen.
Ist ein Klasse oder Funktion innerhalb eines Namespaces deklariert, muss man lediglich darauf achten, das Element mit dem vollqualifizierten Namen aufzurufen:

    [name_of_namespace]::[name_of_member]

Ein Beispiel für die Definition von Namespaces, sowie die richtige Verwendung des vollqualifizierten Namens ist in Abbildung 18 zu sehen.

8 Using-Deklaration und -Direktive

8.1 Using-Deklaration

Mit der using-Deklaration kann man festlegen, auf welchen namespace sich der Aufruf eines bestimmten Elements bezieht.
Dies funktioniert mit:
*1cm using [name_of_namespace]::[name_of_member]

Bei jedem weiteren Aufruf von name_of_member muss anstatt des vollqualifizierten Namens nur noch name_of_member geschrieben werden.

8.2 Using-Direktive

Wenn viele Elemente in einem Namespace deklariert sind, ist die Verwendung der using-Deklaration oft nicht ausreichend. Bei Verwendung einer using-Direktive wird immer die Deklaration aus dem angegebenen Namensbereich benutzt, wenn keine Deklaration im aktuellen scope existiert.
Mit dem Befehl

    using namespace [name_of_namespace]

lässt sich die using-Direktive benutzen.
Die using-Direktive ist ein mächtiger Befehl, deshalb sollte man sie möglichst nur für weit verbreitete namespaces wie den Standard-namespace verwenden, da sonst der Code unübersichtlich wird und seine Funktion nicht gleich ersichtlich ist. Man sollte lieber so oft wie möglich auch die Verwendung der using-Deklaration zurück greifen.

9 Main-Funktion

Die main-Funktion existiert in jedem Programm, da es durch sie erst "gestartet" wird. Sie wird immer zuerst aufgerufen. Ansonsten ist es eine ganz normale Funktion, es können also auch Parameter über- und zurückgegeben werden.
Da die main-Funktion aber als Erstes aufgerufen wird, müssen die Funktionsparameter von außen übergeben werden. Man kann eine unterschiedlich Anzahl von Parametern an die main-Funktion übergeben. Der Funktionskopf sieht folgendermaßen aus:

    int main (int argc, char **argv)

Argc vom Typ Integer und stellt die Anzahl der übergebenen Parameter dar. Der Programmname selbst ist immer der 1. Parameter, deshalb hat argc immer einen Wert größer oder gleich 1. **argv enthält die Werte der Parameter in einer Reihung.
Die main-Funktion sollte immer einen int-Wert zurückgeben, denn beim Verwenden des Programmes in einer Batch-Datei deutet die 0 auf ein erfolgreiches Beenden des Programmes hin, jeder andere Wert jedoch auf einen Fehler.
Der int-Wert wird mit dem return-Befehl in der main-Funktion zurückgegeben.

10 Literaturverzeichnis

[1] Stroustroup, Bjarne. Einführung in die Programmierung mit C++. München: Pearson Studium, 2010.

Abbildungsverzeichnis

```
int i;
static int j = 0;
extern double k;
```

```
double foo();
double foo(double a, double b);
double foo(const int& a);
```

Abbildung 1: Deklarationen

```
extern int i = 5;
```

Abbildung 2: Initialisierung

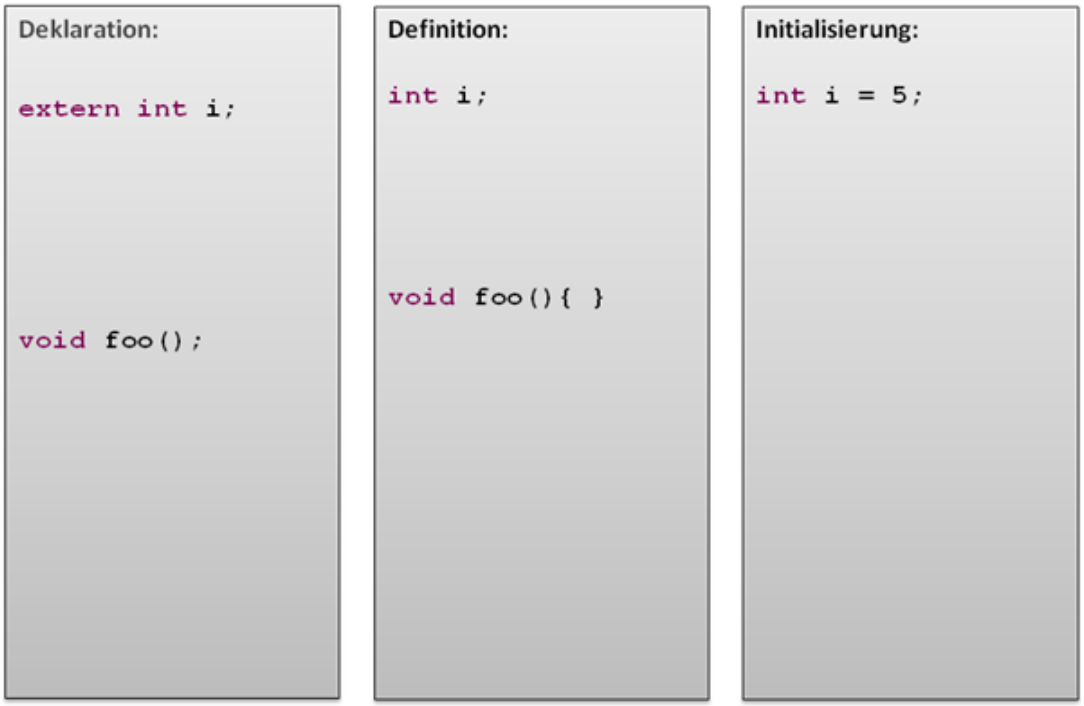


Abbildung 3: Zusammenfassung Deklaration, Definition, Initialisierung

```
void checkRunning(bool isRunning){
   if (isRunning) {
       return;
   } else {
       std::cout << "is not
       running" << std::endl;
   }
}
```

```
int getRandomNumber() {
    return 4;
}
```

Abbildung 4: Rückgabewerte

```
int find (string text, int value) {
    int position = 0;
    //...
    return position;
}
```

```
int find (string text, int value, int hint) {
    int position = 0;
    //...
    return position;
}
```

```
find("15845985", 9); // Aufruf obere Funktion
find("15845985", 9, 5); // Aufruf untere Funktion
```

Abbildung 5: Überladen von Funktionen

```
double find (int value) {
    double position = 0D;
    //...
    return position;
}
```

```
float find (int value) {
    float position = 0F;
    //...
    return position;
}
```

```
int test = (int)find(9); // Welche Funktion?
```

Abbildung 6: Doppelte Signatur

```
int fib(int value) {
   if (value == 0) {
      return 0;
   } else if (value == 1) {
      return 1;
   } else {
      return fib(value - 1) + fib(value - 2);
   }
}
```

Abbildung 7: Pass-by-value

```
int i = 5;
int& k = i;

i++;
k++;

std::cout << i << " == " << k << std::endl;
//7 == 7
```

Abbildung 8: Pass-by-reference

```
double meanRedValue(const Mat& image) {
    double summe = 0.0;
    for (int i = 0; i < image.cols; i++) {
        for (int j = 0; j < image.rows; j++) {
            summe += image.at(i, j)[0];
        }
    }

    return summe / (image.cols * image.rows);
}
//Mat steht für eine Matrix, die ein Array des
//Farbraumes beinhaltet (RGB)
```

Abbildung 9: Pass-by-const-reference

```cpp
#include "std_lib_facilities.h";

using namespace std;

int main() {
    double a = sqrt(4);
    cout << a;
}
//Ausgabe: 2
```

```cpp
using namespace std;

int main() {
    double a = sqrt(4);
    cout << a;
}
//error C3861: "sqrt": Bezeichner wurde nicht gefunden
//error C2065: "cout": nichtdeklarierter Bezeichner
```

Abbildung 10: Verwendung einer Headerdateiund Weglassen des Headers

```cpp
class MyClass1 {
public:
    void f() {
        cout << "Hallo!";
    }
};

class MyClass2 {
public:
    void f() {
        cout << "Hallo!";
    }
};
//beide Methoden können den gleichen Namen haben, da
//sie in unterschiedlichen  Klassen definiert sind
```

Abbildung 11: Gültigkeit von Funktionen im Klassenbereich

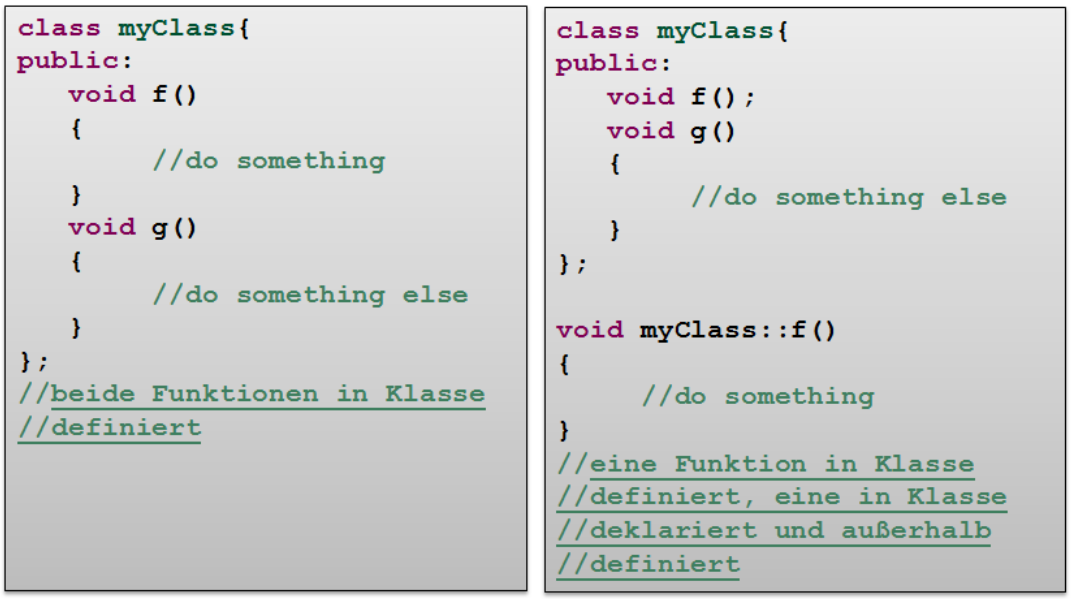
```
bool a = true;
    if (a) {
        int x = 4;
        cout << x;
    } else {
        int x = 5;
        cout << x;
    }

    x = x + 1;
    cout << x;
// error C2065: 'x': nichtdeklarierter Bezeichner
```

Abbildung 12: Deklaration im lokalen Gültigekitsbereich und die Folgen

```
for (int i = 1; i < 10; i++) {
   cout << "i: " << i << " \t";
}
i = 4;
// error C2065: 'i': nichtdeklarierter Bezeichner
```

Abbildung 13: Deklaration im Anweisungsbereich und die Folgen



```
class myClass{
public:
   void f()
   {
        //do something
   }
   void g()
   {
        //do something else
   }
};
//beide Funktionen in Klasse
//definiert
```

```
class myClass{
public:
   void f();
   void g()
   {
        //do something else
   }
};

void myClass::f()
{
     //do something
}
//eine Funktion in Klasse
//definiert, eine in Klasse
//deklariert und außerhalb
//definiert
```

Abbildung 14: Memberfunktionen

```
class MyClass {
    class MyMemberClass {
        class AnotherMemberClass {
            //...
        }
        //...
    }
    //...
}
```

Abbildung 15: Verwenden von Memberklassen

```
void f() {
    //...
    class MyClass {
        //...
    };
    //...
    class MyClass2 {
        //...
    };
    //...
}
```

Abbildung 16: Verwenden von lokale Klassen

```
void f() {
    //...
    void g() {
        //...
    }
    //...
}
// error C2601: 'g': Lokale Funktionsdefinitionen sind
// unzulässig
```

Abbildung 17: Eingebette Funktion

```
namespace Namespace1 {
   namespace Namespace2 {
        void f() {
           cout << "Hallo!";
        }
    }
}

void main() {
    f();
}
// error C3861: 'f':
// Bezeichner wurde nicht
// gefunden.
```

```
namespace Namespace1 {
    namespace Namespace2 {
        void f() {
            cout << "Hallo!";
        }
    }
}

void main() {
    Namespace1::Namespace2::f();
}
//Ausgabe: Hallo!
```

Abbildung 18: Verwenden von Namespaces