Sichtbarkeit und Lebensdauer

Speicherklassen

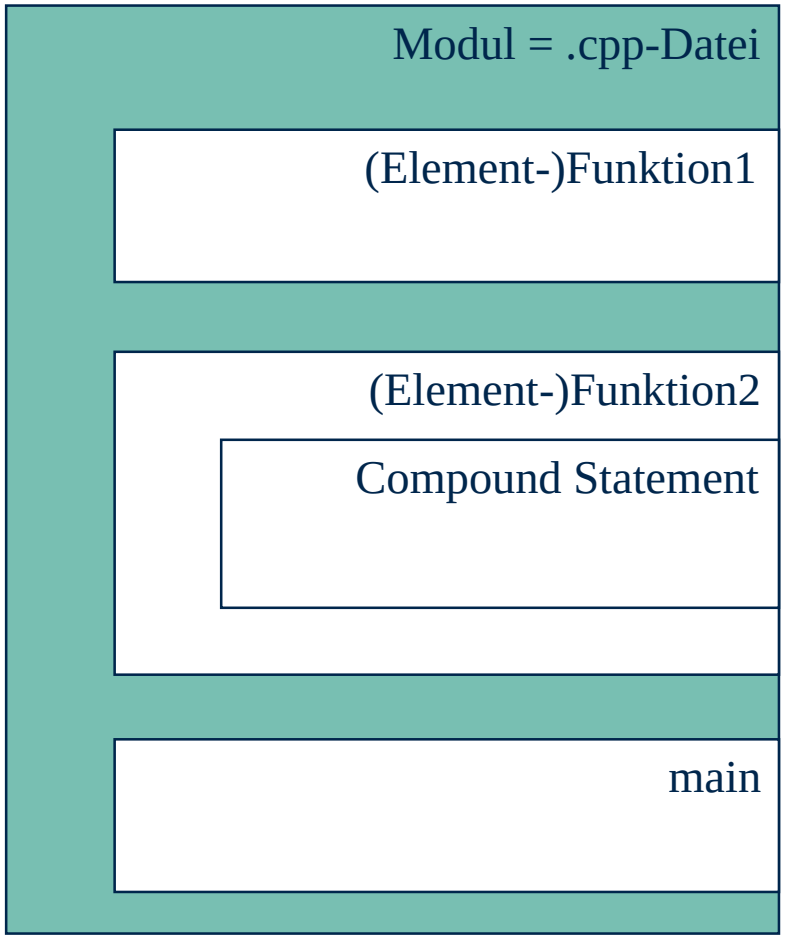
Inhalt

- Geltungsbereich, Sichtbarkeit und Lebensdauer
  - Wo definiere ich Variablen?
  - Wie kann ich Variablen verwenden, die an anderer Stelle definiert sind
  - Wie lange sind solche Variablen gültig
- Wie strukturiere ich meine „Programme“ d.h. „Sourcen“
  - Wann verwende ich .h (Header) Dateien
  - Wie sorge ich dafür, dass diese Dateien auch nur (einmale) verwendet werden
- Wo und wie helfen Namesräume?
- Was bedeuten Schlüsselworte wie:
  - extern
  - Oder static und auto
- Achtung: Ein relativ unübersichtlichsten Kapitel in C/C++
  - durch Unsauberkeit in der Begriffsbildung
  - zum Trost: objektorientiert ist die Sache viel einfacher !

Block- und Modulstruktur

Begriffe

- Modul
  - entspricht einer Quelltextdatei
  - ist "Übersetzungseinheit"
  - enthält i.a. mehrere Blöcke
- Block
  - Hauptprogramm main
  - (Element-)Funktion, Methode
  - Compound Statement
- Deklaration ortsabhängig:
  - global auf Modulebene
  - lokal innerhalb eines Blocks (auch formale Parameter)

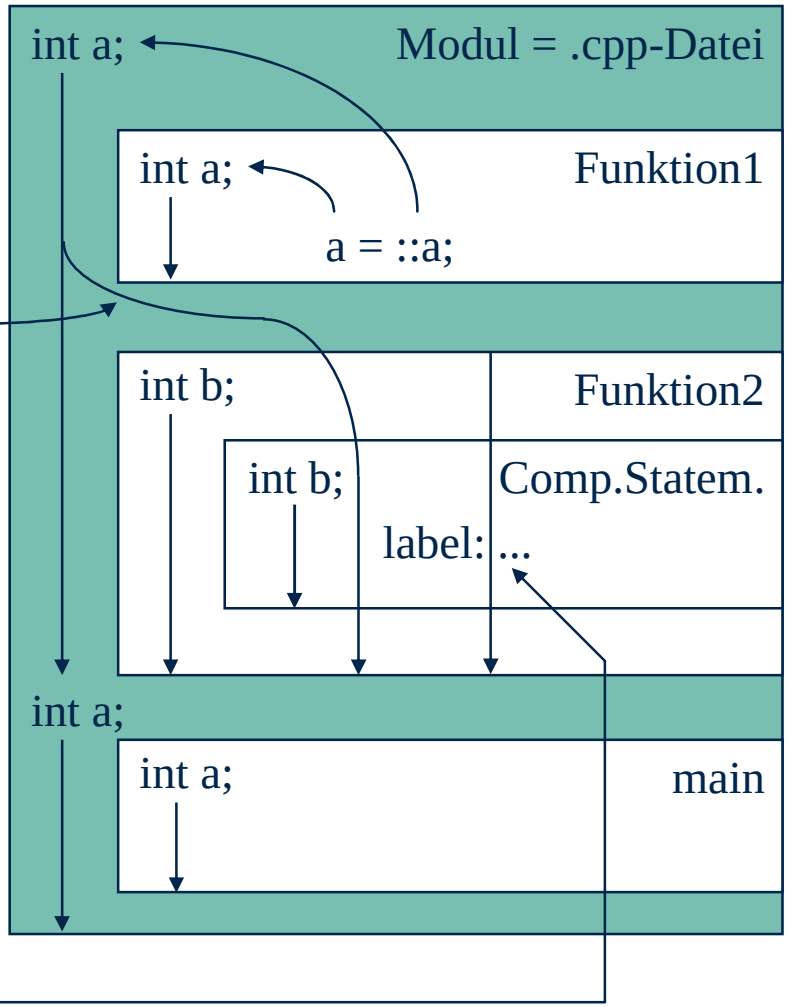


Geltungsbereich von Namen

vom Compiler verwaltet

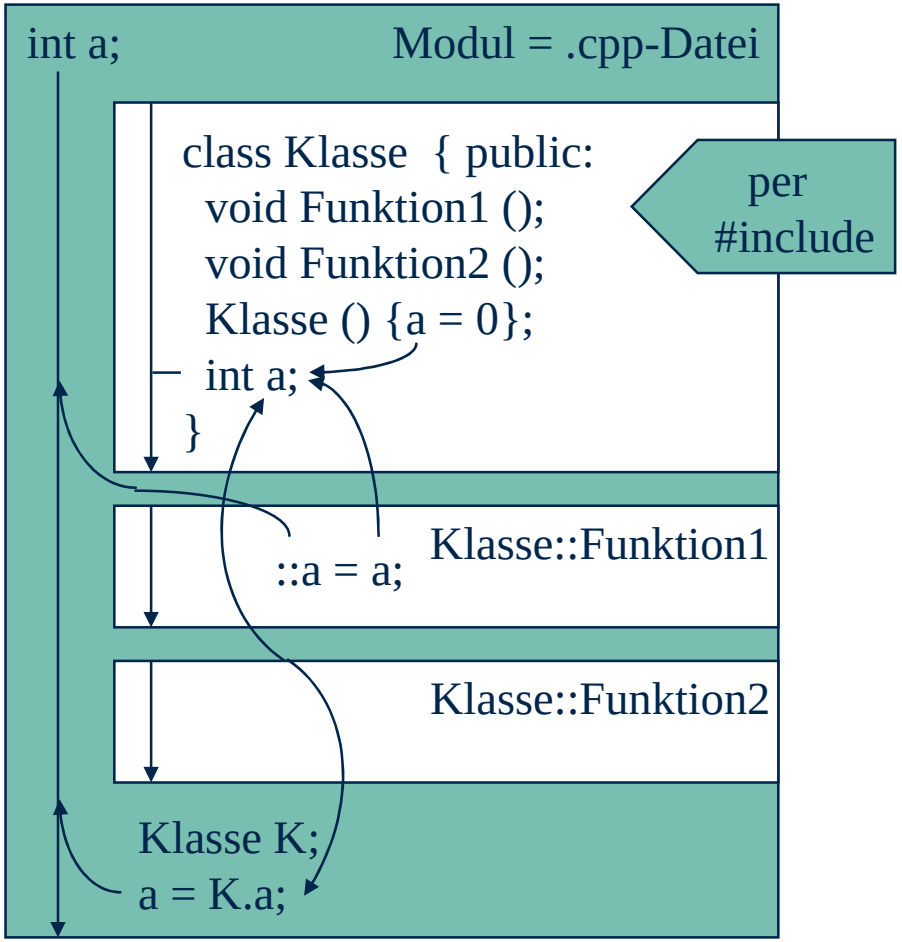
scope ::

- Geltungsbereich (Bezugs-rahmen) von Deklaration bis
  - global: Ende der Datei
  - lokal: Ende des Blocks
- auch in geschachtelten Blöcken
  - sofern nicht redeklariert
  - innere Deklaration verbirgt die äußere
  - Zugriff auf Global mit ::
- mehrfache Deklaration zulässig
  - muß aber identisch sein
- Sonderfall Sprungmarke
  - gesamte umschließende Funktion



Geltungsbereich von Klassenelement-Namen

- Geltungsbereich überdeckt
  - gesamte Klassendeklaration (auch vor dem Ort der Deklaration)
  - jede ausgelagerte Definition einer Elementfunktion
- innerhalb des Geltungsbereichs werden globale Deklarationen gleichen Namens verborgen
- Zugriff von außen mit Klassenname::Elementname



Oder bei Verwendung von Bibliotheken

Probleme bei größeren Entwicklungen

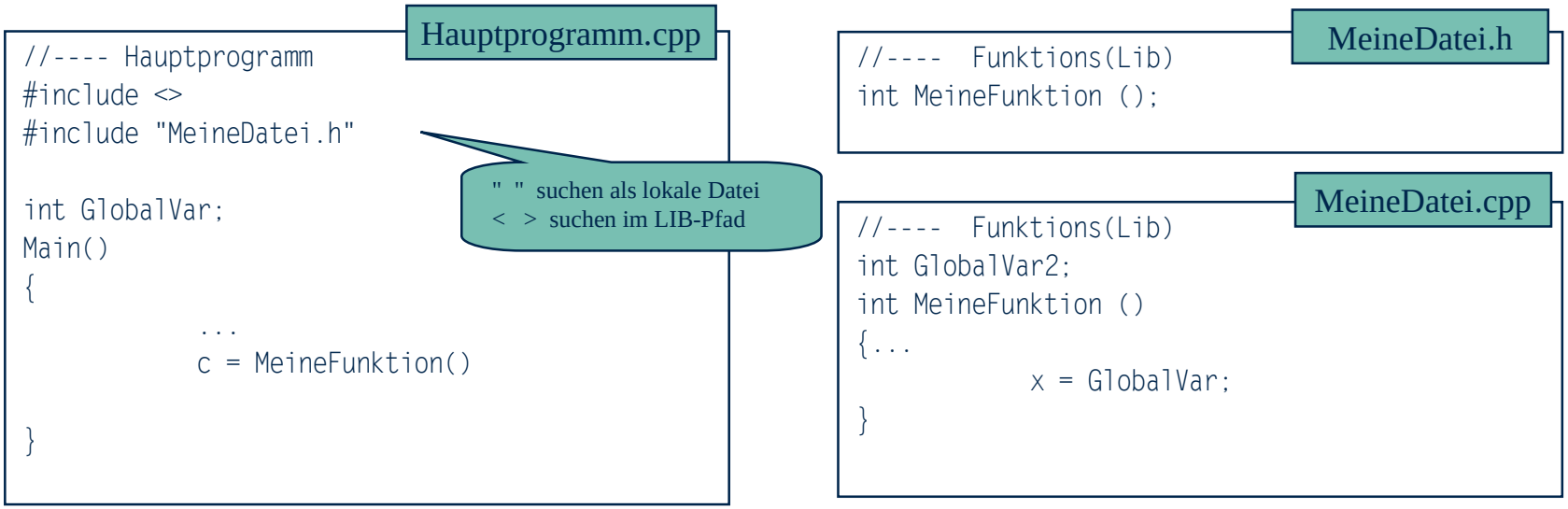
- Wie sage ich meinem Hauptprogramm, (wie) und ob MeineFunktion() definiert ist?
  - Header Datei
- Wie kann ich die Funktion in meinem Program „dem .EXE“ verwenden?
  - Linker
- Was bedeutet Globale Variable für Deklarationen in unterschiedlichen Dateien?
  - Kann ich GlobalVar in MeineFunktion() verwenden?
  - Und ist das eine gute Idee?
- Wie sorge ich dafür, dass meine Variablen-Namen „eindeutig“ sind?

```
//---- Hauptprogramm
#include <>
int GlobalVar;
Main()
{
          ...
          c = MeineFunktion()

}
```

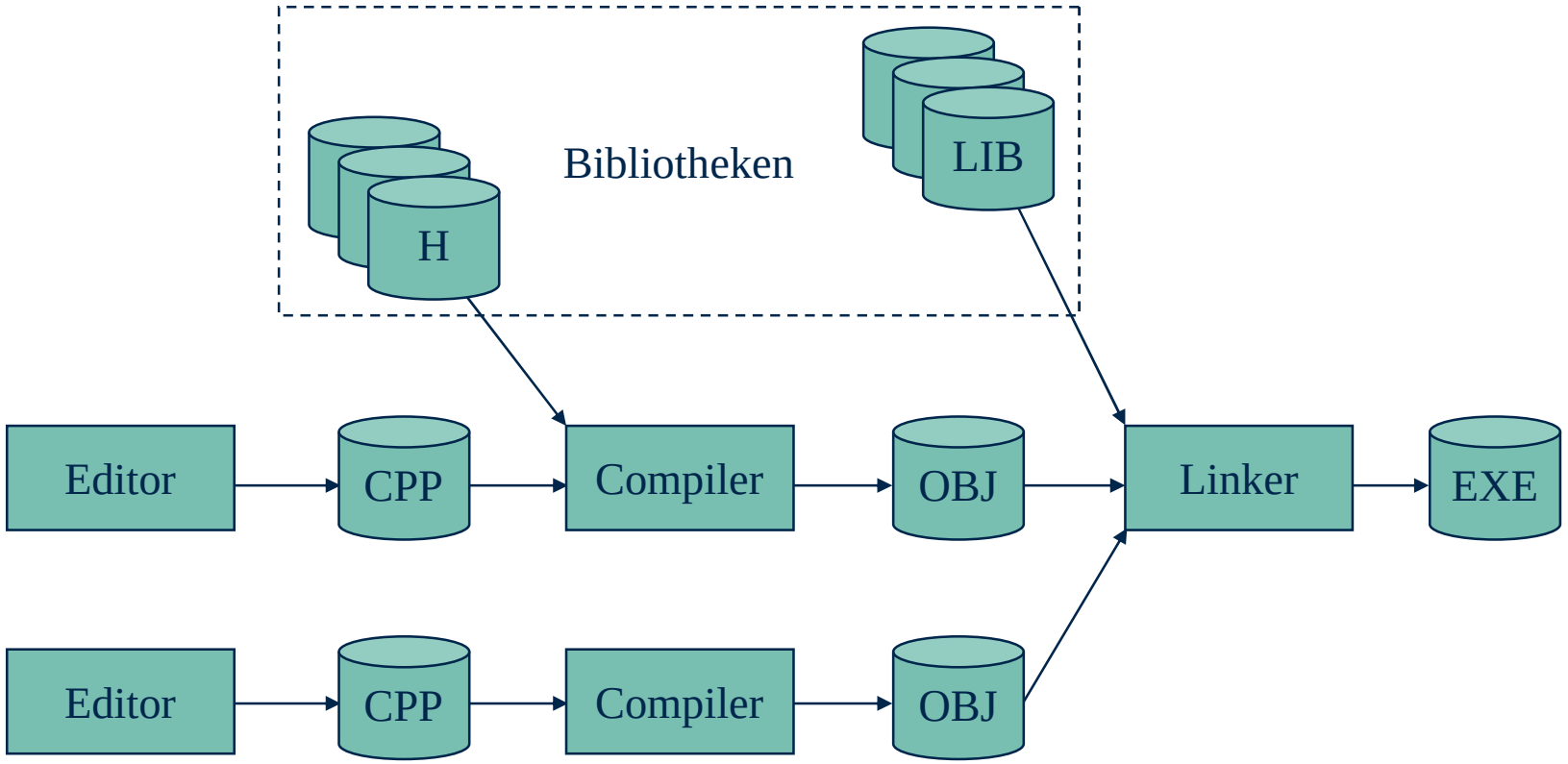
```
//----  Funktions(Lib)
int GlobalVar2;
int MeineFunktion ()
{...
          x = GlobalVar;
}
```

Was braucht ein Compiler zum Übersetzen

- Beim Übersetzen jeder Quelldatei (Source) werden alle globalen Symbole
  - Klassen und Funktionen
  - Globale Variablen (die außerhalb eines Blocks definiert sind)
  - dem gesamten Programm (d.h. global) zur Verfügung gestellt.
- Wie weiß aber meine erste Quelldatei, wie die Funktion in der zweiten Quelldatei definiert ist?
  - Prototyp oder Vorwärtsdeklaration

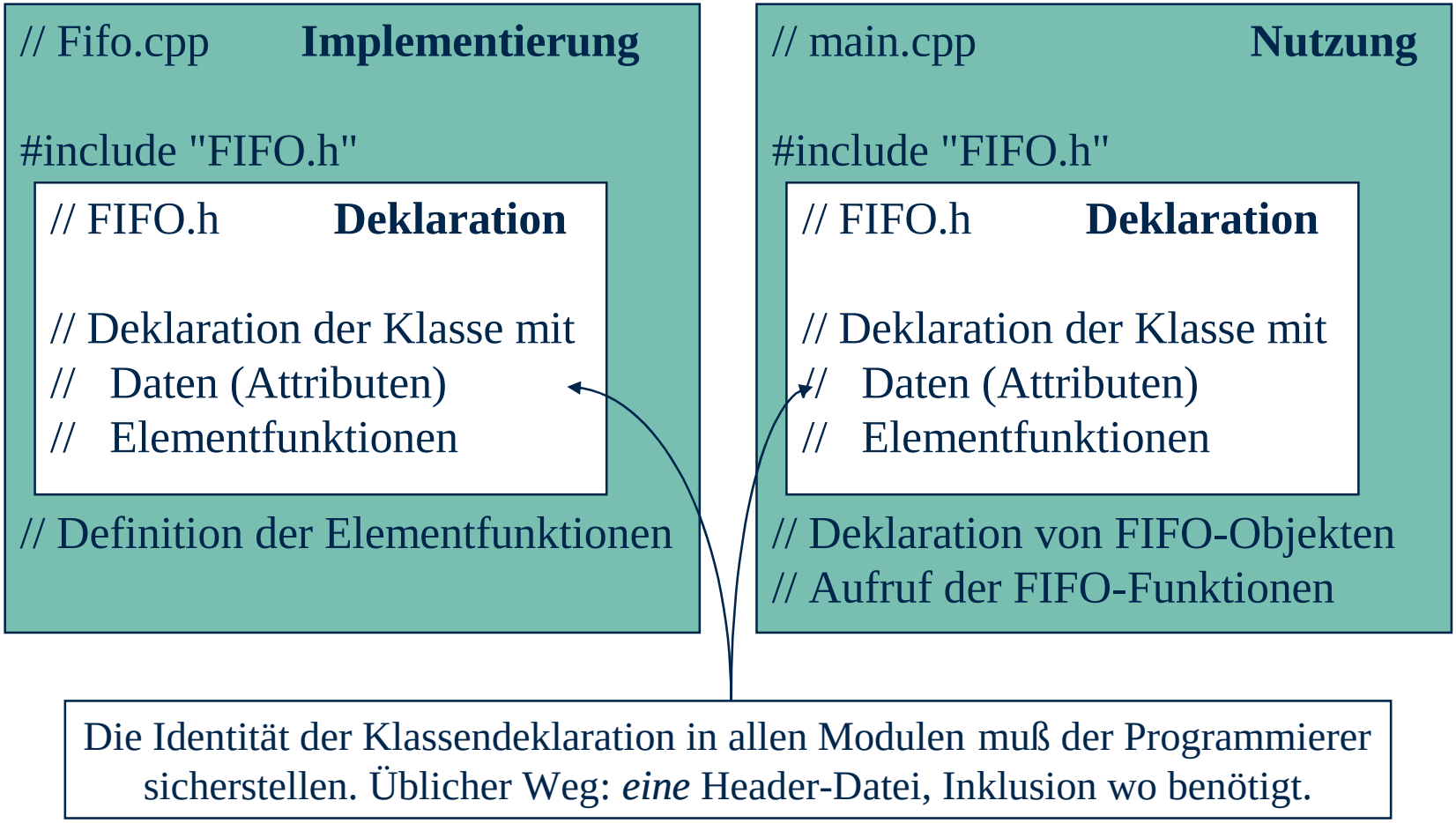


Rückblick: "Der Weg zum EXE"



Definition und Nutzung von Klassen

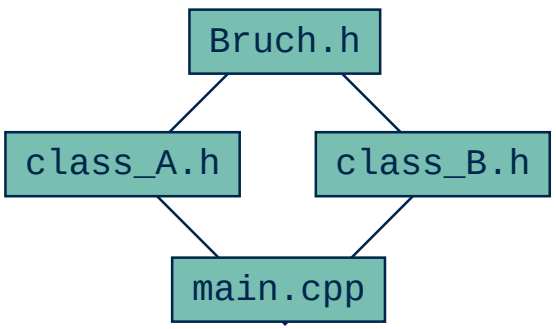
Recht einfach, solange man streng objektorientiert nur Klassen exportiert

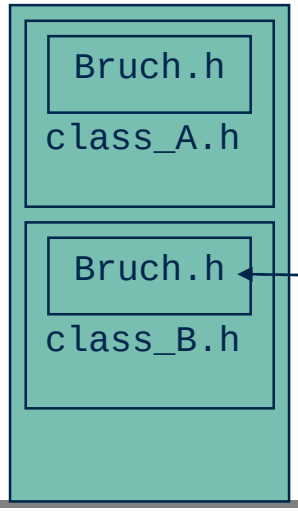


Die Identität der Klassendeklaration in allen Modulen muß der Programmierer sicherstellen. Üblicher Weg: *eine* Header-Datei, Inklusion wo benötigt.

Vermeidung mehrfacher Inklusion

- Prinzip: jede Datei inkludiert nur das, was sie selber braucht
  - kein #include für inkludierte
- mehrfache Inklusion kann auftreten wenn verschiedene .h-Dateien dieselben Deklarationen benötigen und deren .h-Datei inkludieren:





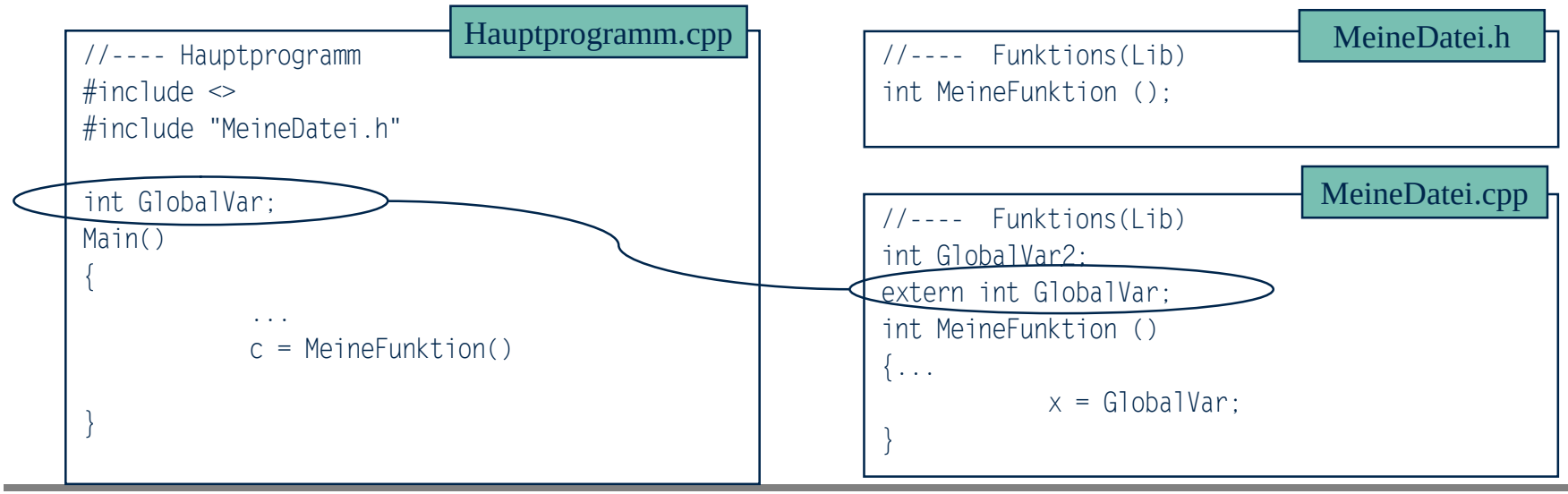
Abhilfe:

```
// Header-Datei Bruch.h
#ifndef BRUCH_H
#define BRUCH_H

class CBruch {
      int Zaehler, Nenner;
    public:
      ...
    };

#endif
```

wird beim 2. Mal übersprungen

One Source – nur eine Stelle der Definition

- Eine Funktion darf mehrfach "vordefiniert" werden aber nur an EINER Stelle "implementiert" werden
  - Es gibt nur eine Datei mit dem eigentlichem Funktionscode
- Eine Globale Variable darf auch nur EINMAL deklariert werden
  - Nur an einer Stelle wird Speicherplatz reserviert
  - Die anderen Module verwenden die "anderweitig" oder "extern" definierte Variable



- **Auch Variablen (wie schon Funktionen, oder Klassen,...) können Modulübergreifend (Dateiübergreifend) verwendet werden.**

- **externe** Elemente ***Import** aus anderem Modul*
  - Nutzung von Elementen eines anderen Moduls

- öffentliche (**public**) Elemente ***Export** für anderes Modul*
  - werden anderen Modulen zur Verfügung gestellt

Anmerkung zu globalen Elementen

- Die Verwendung globaler Variablen ist immer gefährlich und sollte vermieden werden
  - Seiteneffekte, keine Datenkapselung
- eine objektorientierte Anwendung sollte *höchstens ein* globales Objekt haben
  - mit MFC typischerweise das Objekt der Applikationsklasse
  - viele globale Objekte zeugen meist von zu wenig Überlegung bezüglich der Zuordnung der Objekte untereinander
- eine objektorientierte Anwendung sollte *möglichst wenige freistehende* Funktionen haben
  - Ausnahmen sind main und gewisse zweistellige Operatoren
  - man überlege sich, welchen Klassen die Funktionen sinnvollerweise zugeordnet werden können

Linken von C und C++ Modulen

- Linker kennt von Funktionen
  - in C nur den Namen
  - in C++ Namen und Parameter
- dadurch kann der Linker in C++ die Übereinstimmung der Parameterliste zwischen Defi-nition und Aufruf überwachen
- C++ Module können C-Funktionen aufrufen
  - nicht umgekehrt !

überladener Funktionsname

```
// C++ Modul:
int Funktion  (double a, char* b)
int Funktion  (int a, char* b)
void main ()
```

```
/* C Modul: */
int C_Funktion (double a, char* b)
```

```
Linker-Tabelle:
0001:0000  ?Funktion@@ZAHNPAD@Z
0001:0018  ?Funktion@@ZAHHPAD@Z
0001:0078  _main
0001:0090  _C_Funktion
```

```
extern "C" char* gets (char* s);
extern "C" {
   int getchar (void);
}
```

Verwendung eindeutiger Namen

- *Problem*: Klassen sind meist auf globaler Ebene deklariert
  - dadurch besteht die Gefahr von Namenskonflikten, besonders bei Nutzung von Bibliotheken
- *Lösung*: Deklarationen aller Art können als benannter `namespace` gruppiert werden
- Bezug auf deklarierte Namen mit Namespace-Name::
- Pauschale Öffnung des Namensraums mit using-Deklaration
- Dies gilt nicht nur Für Klassen, sondern auch für alle Variablen,,, die in einem namespace definiert sind.
  - Z.B. Flags bei Ein/Ausgabe ios::state

Die Standard C++ Library ist im `namespace std` deklariert

```
namespace std {
  class string {...};
}

namespace MyLib {
  class string {...};
}

using namespace xyz::Klasse1;

using namespace std;
void main ()
{
  string StandardString;
  MyLib::string MeinString;
  Klasse1 MeineKlasse1;
}
```

Geltungsbereich aller Namen in `std`

- **permanente** Variablen und Klassenobjekte

  für spezielle Funktionalität

  - werden vor Programmstart einmalig initialisiert
  - Speicherplatz bleibt reserviert bis zur Beendigung des Programms
  - üblicherweise im Datensegment allokiert

- **temporäre** Variablen und Klassenobjekte

  sparen Speicherplatz

  - werden bei Aufruf einer Funktion neu angelegt und initialisiert
  - werden bei Beendigung der Funktion aufgegeben
  - haben kein Gedächtnis von einem Funktionsaufruf zum nächsten !
  - üblicherweise auf dem Stack allokiert

Sprachmittel zur Festlegung der Lebensdauer

- permanent — *statisch*
  - alle globalen Variablen
  - lokale Variablen mit Schlüsselwort **`static`**

```
static double akkumulator;
```

- temporär — *automatisch*
  - lokale Variablen mit Schlüsselwort **`auto`**

```
auto int Zaehler;
```

  - lokale Variablen ohne weitere Angaben (Normalfall)

```
int Zaehler;
```

  - formale Parameter von Funktionen

```
void PrintTree (node *pNode)
```

Beispiele zur Lebensdauer von Variablen

```
int random (void)
{
  static unsigned long int next;
  unsigned long int temp;

  next = (next*1103515245)+12345;
  temp = (next/65536)%32768;
  return (unsigned int)temp;
}
```

permanent

temporär

```
int stack[10];
int index=0;

void push (int wert)
{
  stack [index] = wert;
  index ++;
}

int pop (void)
{
  int result;

  index--;
  result = stack [index];
  return result;
}
```

permanent

temporär

temporär

Lebensdauer, Funktions-Instanzen, Reentrancy

- Entstehung mehrerer Instanzen einer Funktion:
  2. Aufruf erfolgt bevor 1. Aufruf beendet wurde
  - rekursive Funktionen
  - Multitasking
- erfordert separaten Variablensatz für jede Instanz
  - temporäre Variablen werden für jede Instanz separat angelegt
  - permanente Variablen sind gemeinsam für alle Instanzen
- Funktionen, die in mehreren Instanzen existieren dürfen, nennt man *reentrant*

Private Variablen (auf Modulebene)

- Static deklarierte Variablen und Funktionen sind "permanent" aber nur sichtbar innerhalb eines Modul (Datei)
  - Private Daten

```
static int stack[10];
static int index=0;

static Funktion( int i);

void push (int wert)
{
  stack [index] = wert;
  index ++;
}

int pop (void)
{
  int result;

  index--;
  result = stack [index];
  return result;
}
static Function ( int i)….
```

Die Mehrfachbedeutung von "static"

- auf Modulebene (global) Aussage über Geltungsbereich
  - ohne `static`: public (dem Linker und damit allen bekannt)
  - mit `static`: privat (im ganzen eigenen Modul verfügbar)

- auf Blockebene (lokal) Aussage über Lebensdauer
  - ohne `static`: temporär
  - mit `static`: permanent