

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 6: Gültigkeitsbereiche

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Bisher bekannt:

- Variable im Hauptprogramm
 - sind im Hauptprogramm gültig.
- Lokale Variable in Funktionen
 - sind nur innerhalb einer Funktion gültig und
 - werden ungültig beim Verlassen der Funktion.

Inhalt

- Lokale und globale Variablen
- Namensräume

Globale Variable

sind **Datendefinitionen außerhalb von Funktionen** (z.B. vor `main()`)

- sie existieren bereits **vor Beginn des Hauptprogramms**,
- sie existieren während der **gesamten Lebensdauer des Programms**,
- sie sind im Hauptprogramm und allen Funktionen **sichtbar**,
wenn sie nicht von lokalen Variablen **verdeckt werden**.

```
#include <iostream>
int x = 10; ← globale Variable x
int main() {
    std::cout << x << std::endl;
    int x = 5; ← lokale Variable x verdeckt ab
    std::cout << x << std::endl; dieser Zeile globale Variable x
    return 0;
}
```

Ausgabe: **10**
5

Lokale Variable

sind **Datendefinitionen innerhalb eines Blocks {}**

- sie existieren **ab ihrer Datendefinition** innerhalb des Blockes,
- sie existieren **bis der Block verlassen wird**,
- sie sind auch **in untergeordneten Blöcken sichtbar**,
wenn sie nicht von lokalen Variablen in diesen Blöcken **verdeckt werden**.

```
#include <iostream>
int main() {
    int x = 1;
    std::cout << x << std::endl;
    { int x = 5;
      std::cout << x << std::endl;
    }
    return 0;
}
```

lokale Variable x

untergeordneter Block:
x **verdeckt** x in main()

Ausgabe: 1
5

Beispiel 1

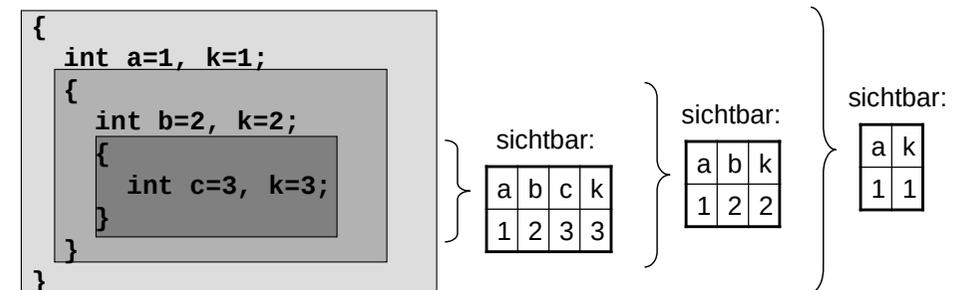
Beispiel 1

```
#include <iostream>
int k = -1;
int main() {
    std::cout << "k global : " << k << std::endl;
    int k = 0;
    std::cout << "k main : " << k << std::endl;
    { int k = 1;
      std::cout << "k block 1: " << k << std::endl;
      { int k = 2;
        std::cout << "k block 2: " << k << std::endl;
      }
      std::cout << "k block 1: " << k << std::endl;
    }
    std::cout << "k main : " << k << std::endl;
    std::cout << "k global : " << ::k << std::endl;
    return 0;
}
```

„scope resolution“

Lokale Variable

- ... verdecken Variable in umgebenden Blöcken, **falls Bezeichner gleich**.
- Verdeckte Variablen sind dann nicht sichtbar, **aber existent!**
- **Unverdeckte Variable** in allen umgebenden Blöcken sind **sichtbar**.



Globale Variable

- können durch lokale Variable verdeckt werden.
- sind überall (selbst wenn verdeckt) über den **Gültigkeitsbereich-Operator** (*scope resolution operator*) `::` zugreifbar.

Der `::` - Operator ermöglicht den Zugriff auf alle global bekannten Objekte.

Achtung:

Globale Variable sollten **grundsätzlich vermieden werden**.

Beispiel 2

```

C:\WINDOWS\system32\cmd.exe - scope2
E:\EINI NEU>scope2
k global : -1
k funct : 99
k funct : 100
k main : 0
k block 1: 1
k funct : 101
  
```

Beispiel 2

```

#include <iostream>
int k = -1; // global
void funct(int k) {
    k += 100;
    std::cout << "k funct : " << k << std::endl;
}
int main() {
    std::cout << "k global : " << k << std::endl;
    funct(k);
    int k = 0;
    funct(k);
    std::cout << "k main : " << k << std::endl;
    { int k = 1;
      std::cout << "k block 1: " << k << std::endl;
      funct(k);
    }
    return 0;
}
  
```

Beispiel 3

```

#include <iostream>
int k = -1; // global
void funct(int x) {
    x += 100;
    std::cout << "x funct : " << x << std::endl;
    std::cout << "k funct : " << k << std::endl;
}
int main() {
    std::cout << "k global : " << k << std::endl;
    funct(k);
    int k = 0;
    funct(k);
    std::cout << "k main : " << k << std::endl;
    return 0;
}
  
```

An arrow points from the word "global" in a circle to the `k` variable in the `std::cout << "k funct : " << k << std::endl;` line of the `funct` function.

Beispiel 3

```

C:\WINDOWS\system32\cmd.exe - scope3
E:\EINI NEU>scope3
k global : -1
x funct : 99
k funct : -1
x funct : 100
k funct : -1
k main : 0

```

Beispiel 5

Zählvariable wird nach der Schleife noch benötigt?
⇒ Datendefinition von k **vor** dem Schleifenkopf

```

int k; // ab hier existiert k
for (k = 0; k < n; ++k)
    sum += k;
std::cout << k << std::endl; // k existiert noch

```

Zählvariable wird nicht mehr benötigt?
⇒ Datendefinition von k **im** Schleifenkopf

```

for (int k = 0; k < n; ++k) // ab hier existiert k
    sum += k;
// k existiert hier nicht mehr
// Die folgende Zeile würde nicht compilieren:
//std::cout << k << std::endl;

```

→ Fehlermeldung z.B.
error: 'k' was not declared in this scope

Beispiel 4

```

#include <iostream>
int main() {
    int i, sum = 0;
    for (i = 0; i < 3; i++) {
        int j; ← Datendefinition im inneren Block!
        for (j = 0; j < 4; j++) {
            sum += i * j;
            std::cout << sum << std::endl;
        }
    }
    return 0;
}

```

Merke:

In **jedem Block** dürfen neue lokale Variable angelegt werden.
Sie verdecken Variable gleichen Namens in äußeren Blöcken.

Statische (globale) Variable

sind globale Variable, die nur in der Datei sichtbar sind, in der sie deklariert werden.

Datendefinition:

static Datentyp Bezeichner;

Dateninitialisierung:

static Datentyp Bezeichner = Wert;

```

#include <iostream>
int global = 1; ← globale Variable für alle Dateien
static int statisch = 2; ← globale Variable nur für diese Datei

```

Datei *Global.cpp*

```

int main() {
    cout << global << endl;
    cout << statisch << endl;
    return 0;
}

```

Datei *Haupt.cpp*



Fehler!

Sowohl **global** als auch **statisch** nicht sichtbar!

?

```
#include <iostream>
int global = 1;
static int statisch = 2;
```

Datei *Global.cpp***2. Versuch:**

```
int global;
int statisch = 3;
int main() {
    cout << global << endl;
    cout << statisch << endl;
    return 0;
}
```

Datei *Haupt.cpp*

Frage: Wie kommt man an die globalen Variablen, die in **anderen** Dateien definiert worden sind?

Idee: Variable müssen vor ihrem ersten Gebrauch definiert worden sein!

Fehler!

Der Linker meldet, dass Variable **global** bereits in *Global.cpp* definiert worden ist.

Nicht-statische globale Variable sind **in allen Dateien** globale Variable!

Hier: Versuch, **erneut** globale Variable gleichen Namens zu definieren.

```
#include <iostream>
int global = 1;
static int statisch = 2;
```

Datei *Global.cpp***3. Versuch:**

```
extern int global;
int statisch = 3;
int main() {
    cout << global << endl;
    cout << statisch << endl;
    return 0;
}
```

Datei *Haupt.cpp*

Frage: Wie kommt man an die globalen Variablen, die in **anderen** Dateien definiert worden sind?

Idee: Durch Schlüsselwort **extern** angeben, dass Variable **global** außerhalb dieser Datei definiert ist.

Keine Fehlermeldung!

Aufruf des Programms liefert Ausgabe:

```
1
3 } ?
```

Zugriff auf **global** → OK

Mit **int statisch** wurde nicht-statische globale Variable deklariert

```
#include <iostream>
int global = 1;
static int statisch = 2;
```

Datei *Global.cpp***4. Versuch:**

```
extern int global;
extern int statisch = 3;
int main() {
    cout << global << endl;
    cout << statisch << endl;
    return 0;
}
```

Datei *Haupt.cpp*

Frage: Wie kommt man an die globalen Variablen, die in **anderen** Dateien definiert worden sind?

Idee: Wenn **extern** bei **global** hilft, dann hilft es vielleicht auch bei **statisch**? (Hmm, schwache Idee ...)

Fehler!

Linker meldet, dass das externe Symbol **int statisch** nicht aufgelöst werden konnte.

⇒ **Stimmt!** Die Variable **statisch** in der Datei *global.cpp* ist eine **statische** Variable und nur dort gültig!

Fazit: Man kann nicht aus anderen Dateien auf statische globale Variable zugreifen.

Achtung:

Statische globale Variable sind Erbstück aus C.

Sie gelten in C++ als **überholt** (*deprecated*).

Zukünftige Versionen von C++ könnten das nicht mehr unterstützen!

⇒ **Nicht verwenden!**

Nicht verwechseln:

Statische lokale Variable in Funktionen sind auch Erbstück aus C.

Sie sind in C++ willkommen! →

Statische Variable (in Funktionen)

haben einen **anderen Gültigkeitsbereich** als „normale“ Variablen.
Eine statische Variable **in einer Funktion** hört nicht auf zu existieren, wenn die Funktion beendet wird, sondern **bleibt im Speicher bestehen**.

Achtung: Hat gleichen Sichtbarkeitsbereich wie normale lokale Variablen!

```
unsigned int countCalls() {
    static unsigned int ctr = 0;
    return ++ctr;
}
int main() {
    for (int i = 0; i < 10; i++)
        cout << countCalls() << endl;
    return 0;
}
```

Statische (lokale) Variable werden nur **einmal initialisiert**, nämlich beim 1. Aufruf der Funktion.

Sie bleiben gültig bis zum Ende des gesamten Programms; also über das Ende der Funktion hinaus.

Die Zeile **static unsigned int ctr=0** wird somit nur **einmal** ausgeführt. Die statische lokale Variable **ctr** behält ihren Wert bei weiteren Funktionsaufrufen.

Ausgabe: Zahlen 1 bis 10

Namensräume (namespaces)

- eingeführt mit dem ersten **C++-ISO-Standard** von 1998
- zur **Vermeidung von Namenskonflikten** bei großen Programmen mit vielen Entwicklern

```
void drucke(int n, double a[]){
    double sum = 0.0;
    while (--n >= 0) sum+=a[n];
    std::cout << sum;
}
```

Entwickler A

```
void drucke(int n, double a[]){
    for (int i = 0; i < n; i++)
        std::cout << a[i] << ' ';
    std::cout << std::endl;
}
```

Entwickler B

```
int fkt1(int wert) {
    static int w = -1;
    if (wert != 0) w = wert;
    return w;
}
int fkt2(int a) {
    {
        static int b = a;
    }
    //return b;
    return a;
}
int main() {
    cout << fkt1(0) << " " <<
        << fkt1(3) << " " <<
        << fkt1(0) << endl;
    return 0;
}
```

w wird beim 1. Aufruf mit **-1** initialisiert.
w bleibt unverändert, wenn **wert == 0**.
w wird zu **wert**, wenn **wert** ungleich **0**.

statische Variable **b** in neuem Block:
existiert **bis zum Ende des Programms**

würde **Compilerfehler** liefern:
b existiert zwar noch,
aber der Sichtbarkeitsbereich (Block)
wurde bereits verlassen!

Ausgabe: **-1 3 3**

```
namespace A {
    void drucke(int n, double a[]){
        double sum = 0.0;
        while (--n >= 0) sum += a[n];
        std::cout << sum;
    }
}
namespace B {
    void drucke(int n, double a[]){
        for (int i = 0; i < n; i++)
            std::cout << a[i] << ' ';
        std::cout << std::endl;
    }
}
void print_all(int n, double a[])
{
    B::drucke(n, a);
    A::drucke(n, a);
}
```

Auflösung des Namenskonfliktes durch *namespaces*

Namensräume

- können dazu benutzt werden, Funktionen etc. **nach Einsatzgebiet** zu ordnen
- „Wegsperrern“ von **selten benutzten** Funktionen
- bei häufig benutzten Funktionen / Namensräumen kann dann durch **using-**Direktive der qualifizierende Namensteil weggelassen werden

```
A::drucke(n, a);
```

qualifizierender Namensteil =
Name des Namensraums

```
using namespace A;  
drucke(n, a);
```

verwendet **drucke()** aus Namensraum **A**
(dessen Namen nun **alle** im globalen
Namensraum sichtbar sind)

```
using A::drucke;  
drucke(n, a);
```

verwendet **drucke()** aus Namensraum **A**
(aus **A** nur **drucke** sichtbar)

Beispiel 5

funktioniert immer:

```
int k; // ab hier existiert k  
for (k = 0; k < n; ++k)  
    sum += k;  
std::cout << k << std::endl; // k existiert noch
```

bei (sehr) alten Compilern:

```
for (int k = 0; k < n; ++k) // ab hier existiert k  
    sum += k;  
std::cout << k << std::endl; // k existiert noch
```

bei aktuellen Compilern:

```
for (int k = 0; k < n; ++k) // ab hier existiert k  
    sum += k;  
std::cout << k << std::endl; // k existiert nicht mehr
```

→ Fehlermeldung „out of scope“ o.ä.

Namensräume

zu Anfang der Vorlesung:

```
std::cout << a << std::endl;
```

im Namensraum **std** liegen Standardfunktionen (auch aus C)

```
using namespace std; ← dadurch ...  
// ...  
cout << a << endl; ← ... entfällt das lästige std::
```