

Speicherklassen

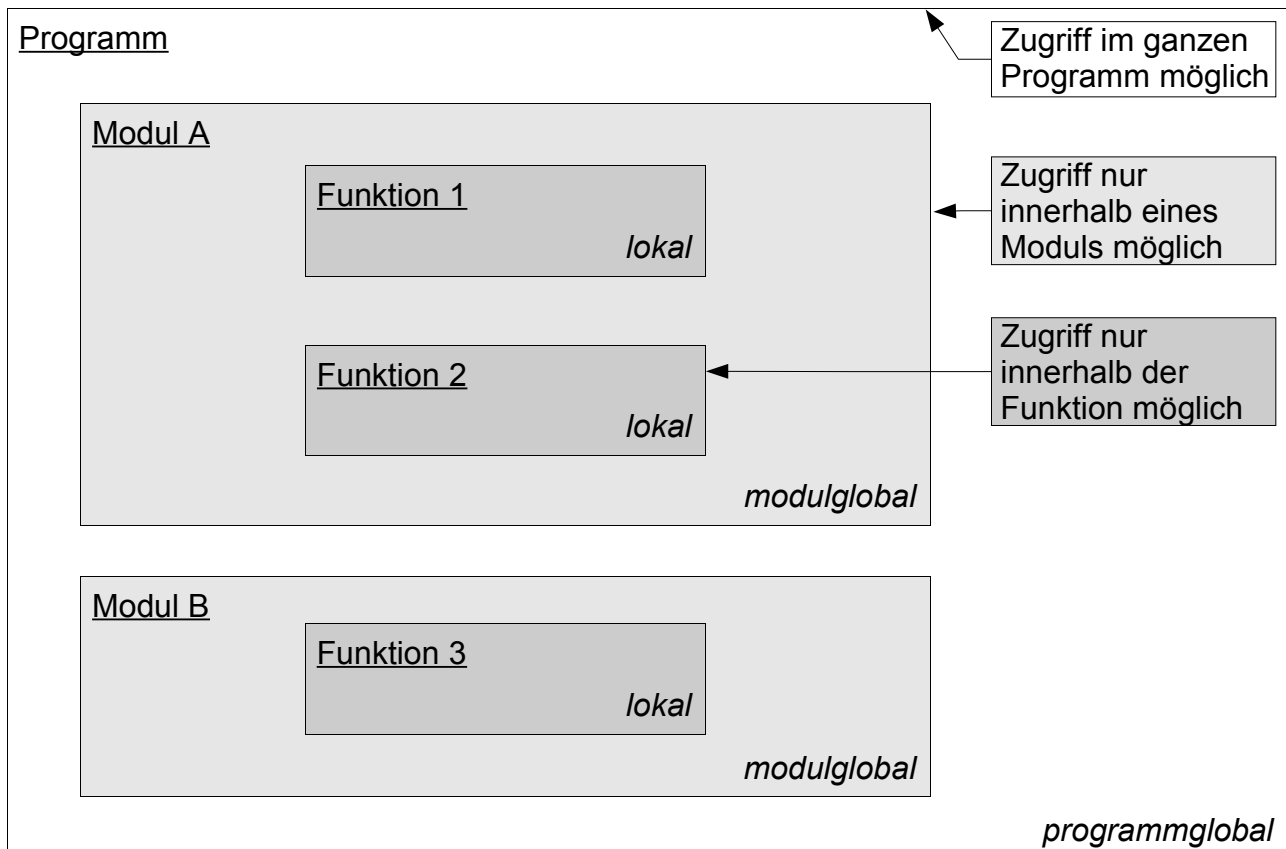
Eine **Speicherklasse** legt fest, in welchen Teilen eines Programmes ein Objekt oder eine Funktion verwendet werden kann.

- Unterscheidung zwischen *Lebensdauer* und *Sichtbarkeit* einer Variable oder Funktion, welche abhängen von "Position der Deklaration" und "Spezifikations"-Kennzeichner

Die **Lebensdauer** definiert die Dauer der Erhaltung einer Variable oder Funktion im Hauptspeicher. Es wird zwischen *permanent* und *temporärer* Lebensdauer unterschieden.

- Festlegungen:
 1. Lebensdauer von Funktionen ist stets *permanent*
 2. Lebensdauer von Variablen hängt von Position der Deklaration ab und ist wie folgt, wenn nicht anders spezifiziert:
 - innerhalb einer Funktion: *temporär*
 - außerhalb einer Funktion: *permanent*

Die **Sichtbarkeit** definiert die aktuelle Zugriffsmöglichkeit auf eine Variable oder Funktion. Es wird zwischen *lokaler*, *modulglobaler* und *programmglobaler* Sichtbarkeit unterschieden.



- Festlegungen:
 1. Sichtbarkeit von Funktionen ist *programmglobal*, wenn nicht anders spezifiziert
 2. Sichtbarkeit von Variablen hängt von Position der Deklaration ab und ist wie folgt, wenn nicht anders spezifiziert:
 - innerhalb einer Funktion: *lokal*
 - außerhalb einer Funktion: *programmglobal*
- Beispiel:

```

#include <iostream>
using namespace std;

int x = 3;

int func()
{
    int y = 5;
    return y;
}

int main()
{
    cout << func() << endl;
    cout << x << endl;
    return 0;
}

```

Spezifikation “static”

Das Schlüsselwort “**static**” kann Variablen- und Funktionsdeklarationen vorangestellt werden, um ihre zugehörige Speicherklasse (Lebensdauer und Sichtbarkeit) zu beeinflussen.

- Beispiel: `static int zaehler;`
`static char lies_zeichen();`
- Festlegungen für Lebensdauer:
 1. Lebensdauer von statischen Funktionen ist stets *permanent*
 2. Lebensdauer von statischen Variablen ist stets *permanent*
- Festlegungen für Sichtbarkeit:
 1. Sichtbarkeit von statischen Funktionen ist stets *modulglobal*
 2. Sichtbarkeit von statischen Variablen hängt von Position der Deklaration ab und ist wie folgt:
 - innerhalb einer Funktion: *lokal*
 - außerhalb einer Funktion: *modulglobal*

- Beispiel:

```

// counter_modul.cpp
static int increase()
{
    static int x = 0;
    x++;
    return x;
}

int counter()
{
    return increase();
}

```

Lebensdauer = *permanent*
 Sichtbarkeit = *modulglobal*

Lebensdauer = *permanent*
 Sichtbarkeit = *lokal*

Lebensdauer = *permanent*
 Sichtbarkeit = *programmglobal*

```

// counter_main.cpp
#include <iostream>
using namespace std;

int counter(); // Funktions-Prototyp

static int geheim = 10;

int main()
{
    for(int i=0; i < geheim; i++)
        cout << counter() << " ";
    cout << endl;
    return 0;
}

```

Lebensdauer = *permanent*
 Sichtbarkeit = *modulglobal*

Lebensdauer = *temporär*
 Sichtbarkeit = *lokal*

- Programm erstellen: `g++ counter_modul.cpp counter_main.cpp -o counter`
- Ausgabe: 1 2 3 4 5 6 7 8 9 10

Spezifikation “extern”

Das Schlüsselwort “**extern**” bezeichnet die *Deklaration* bereits in anderen Quelldateien *definierter* Variablen.

- Wiederholung: eine Variablen-*Definition* reserviert Speicher, eine Variablen-*Deklaration* teilt dem Compiler nur mit, dass diese Variable noch irgendwo definiert ist
- soll eine Variable vor ihrer Definition verwendet werden, so muss (wie in letzter Übung gezeigt) sie zumindest deklariert sein (Typ und Name dann bekannt, Speicher wird später durch Linker bekannt gegeben)

- Beispiel:

```
// div_modul.cpp
#include <iostream>
#include <string>
using namespace std;

// Variablen-Definition
string global_error;

// Funktions-Definition
float division(int x, int y)
{
    if (y==0)
    {
        global_error="ERROR: Division durch Null!";
        return 0;
    }
    return (x/y);
}
```

globaler Fehler-Indikator

Fehler global bekannt machen

```
// div_main.cpp
#include <iostream>
#include <string>
using namespace std;

// Variablen-Deklaration
extern string global_error;

// Funktions-Deklaration
float division(int x, int y);

int main()
{
    float d = division(5,0);
    if (global_error != "")
        cout << global_error << endl;
    else
        cout << d << endl;
}
```

Quasi-Import der Variable global_error

Division durch Null zur Verdeutlichung ausgelöst

- globale Fehlervariable für Funktionen gibt es auch in der Realität:

in der Linux-Konsole eingeben
man errno ↴

Rekursive Funktionen

Eine Funktion ist genau dann **rekursiv**, wenn sie sich selbst aufruft. Eine Rekursion hat stets eine *Abbruchbedingung* und eine *Schrittvorschrift*. Man unterscheidet zwischen *direkter Rekursion* und *indirekter Rekursion*, die über mehrere Stufen entsteht.

- subtile Beispiele:

- direkte Rekursion: *“Dieser Satz ist unwahr”*
- indirekte Rekursion: *“Der folgende Satz ist wahr”*
“Der vorhergehende Satz ist nicht wahr”

- mathematische Beispiele:

- Summe der Zahlen 1 bis n:

$$summe(n) = \begin{cases} 0 & \text{falls } n=0 \\ summe(n-1)+n & \text{falls } n \geq 1 \end{cases} \quad \forall n \geq 0$$

- Fakultät einer Zahl n:

$$fakult(n) = \begin{cases} 1 & \text{falls } n=0 \\ fakult(n-1)*n & \text{falls } n \geq 1 \end{cases} \quad \forall n \geq 0$$

- Programm zur Berechnung der Fakultät einer natürlichen Zahl n:

```
#include <iostream>
#include <iomanip>
using namespace std;

long double fakult(unsigned int n)
{
    if(n<=1)
        return 1; // Rekursionsabbruch
    else
        return ( n * fakult(n-1) ); // Rekursionsschritt
}

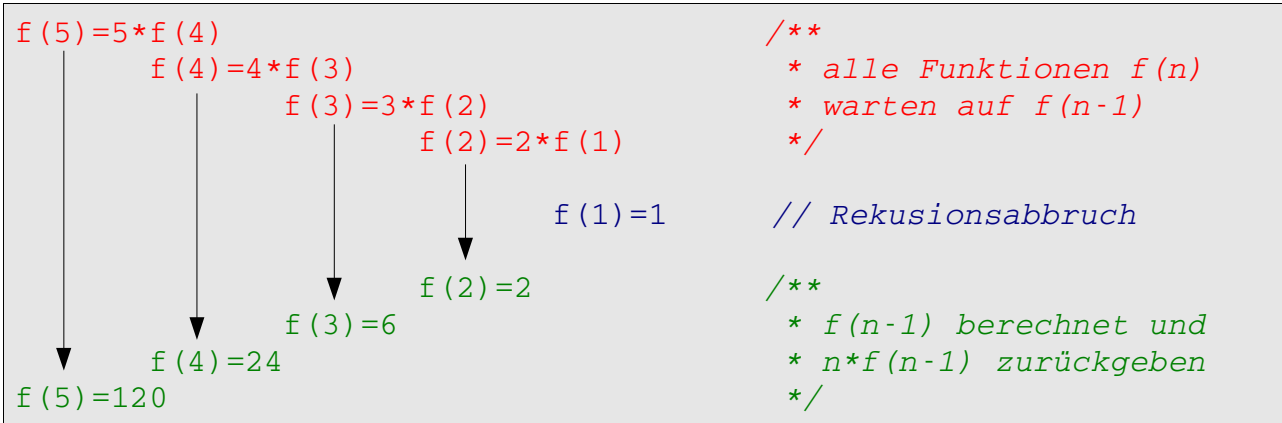
int main()
{
    unsigned int wert=0;

    cout << "Wert eingeben:";
    cin >> wert;
    cout << fixed << fakult(wert) << endl;
    return 0;
}
```

Datentyp für positive ganzzahlige Zahlen

Ausgabe als Gleitpunktzahl erzwingen, da double standardmäßig halblogarithmisch dargestellt wird

- Rekursionsdarstellung für n=5:



Anwendungsbereiche

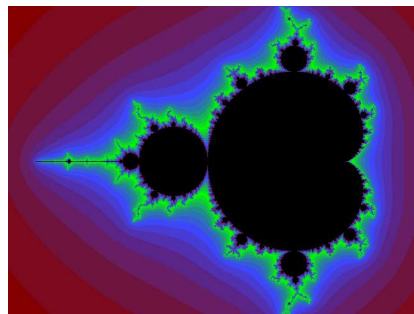
- diverse Probleme der Informatik, wie z.B.:
 - Durchlaufen von Verzeichnisbäumen
 - Quick-Sort-Algorithmus ("Divide and Conquer"-Prinzip)
 - Binäre Bäume zur Verwaltung von Daten
 - Compilerbau (Syntaxanalyse etc.)
 - Grammatiken in Backus-Naur-Form:

```

<Ziffer ausser Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer> ::= 0 | <Ziffer ausser Null>
<Zweistellige Zahl> ::= <Ziffer ausser Null> <Ziffer>
<Zehn bis Neunzehn> ::= 1 <Ziffer>
<Zweiundvierzig> ::= 42
<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>

```

- Mathematik:
 - rekursive Funktionen als alternative "elegantere" Darstellungsform von Berechnungsparadigmen
 - Darstellung komplexer Figuren, wie Fraktale (Mandelbrotbäume):



- Linguistik:
 - Beschreibung komplexer Satzstrukturen

Matrizen – Arbeit mit zweidimensionalen Feldern

Anlegen und Initialisieren

- Allg. Definition einer Matrix:

```
<Element-Datentyp> matrixname [zeilen][spalten];
```

- Beispiel für eine 3x3-Matrix:

```
int matrix[3][3];
```

- Wert eines Matrix-Elementes ändern:

```
matrixname[zeilennummer][spaltennummer] = wert;
```

- Beispiele für eine 3x3-Matrix:

```
matrix[3][3] = 10;  
cin >> matrixname[3][2];
```

- Matrixelemente initialisieren:

Zum sukzessiven Einlesen einer Matrix mit i Zeilen und j Spalten, benötigt man eine 2fach-verschachtelte Schleife. Hier ein Beispiel für eine 9x9-Matrix:

```
for(i=0; i<9; i++)  
    for(j=0; j<9; j++)  
        cin >> matrixname[i][j];
```

Das Initialisieren einer Matrix bei der Definition, ist wie das Initialisieren eines Feldes, wo jedes Element wieder ein Feld ist. Hier ein Beispiel für eine 3x3-Einheits-Matrix:

```
int matrixname[3][3] = { {1,0,0},  
                        {0,1,0},  
                        {0,0,1} };
```