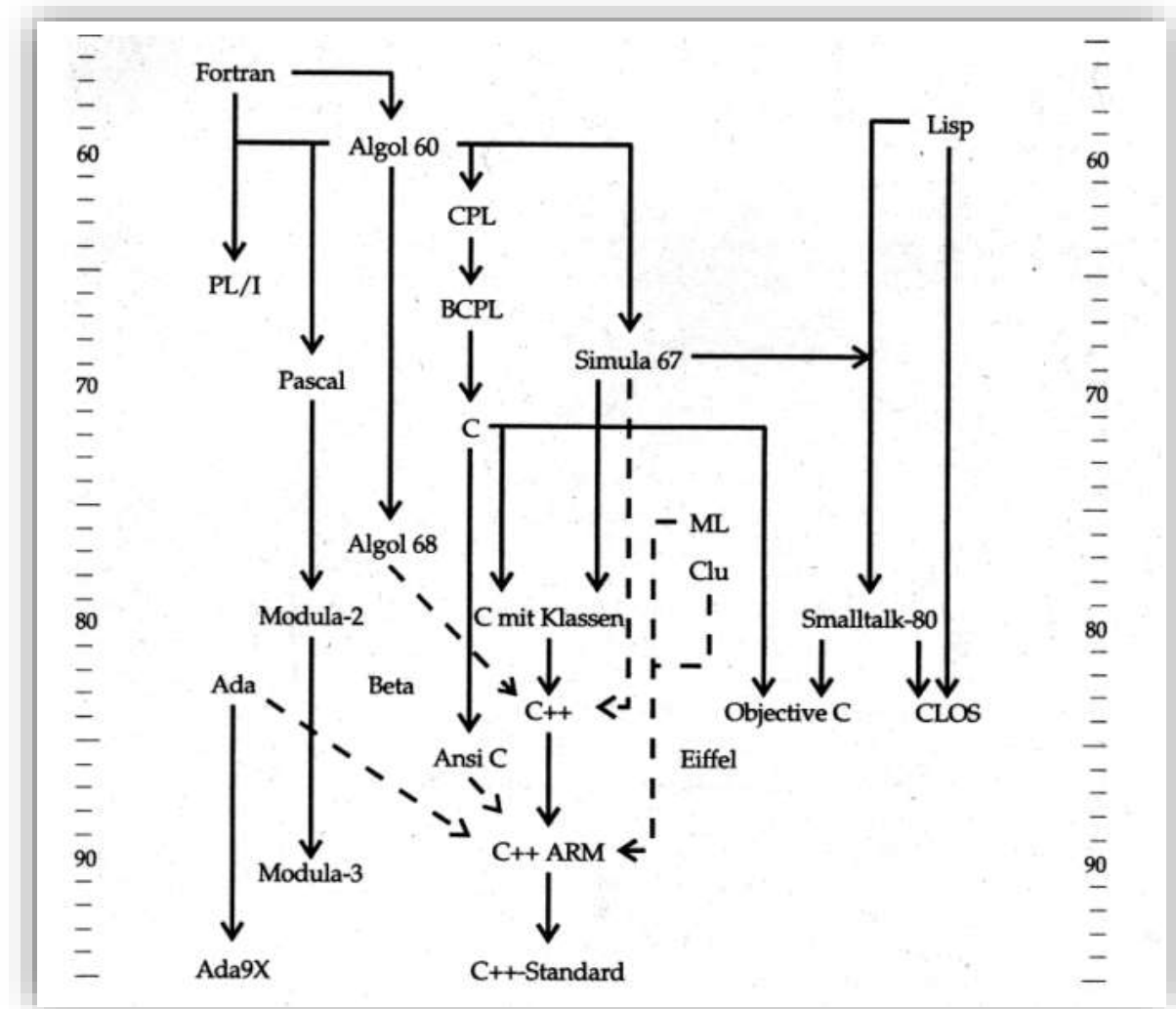


Einführung C++



Historie: Von C zu C++

- Entwicklung gestartet von Bjarne Stroustrup
 - Ursprünglich als Erweiterung von C („C mit Klassen“)
 - Ziel: große Simulationsprojekte mit minimalem Programmieraufwand
- Einflüssen von mehreren Sprachen
 - Simula 67 (Klassenkonzept)
 - Algol 68 (Überladen von Operatoren)
 - BCPL (Kommentare)
 - Ada (Templates)



- Warum C als Basis?
 - Vielseitig, kurz und relativ „low level“
 - Geeignet für viele Aufgaben der Systemprogrammierung
 - Auf allen System und allen Maschinen verfügbar
 - Auf Unix verfügbar – damals ein wichtiges Argument
- Kompatibilität zu C wurde über all die Jahre beibehalten
 - Millionen Zeilen existierender C-Code – können leicht von wenigen C++ Erweiterungen profitieren
 - C war/ist vielen Programmierern bereits bekannt – Minimierung der Hürden C++ zu nutzen
- C ist nach wie vor eine Teilmenge von C++ und wesentliche Grundlage

„C++ wurde eigentlich nur erfunden, um dem Autor und seinen Freunden das Programmieren in Assembler, C oder einer anderen modernen Hochsprache zu ersparen. Es geht in der Hauptsache darum, das Schreiben guter Programme zu erleichtern und das Programmieren für den individuellen Programmierer kurzweiliger zu machen.“

Bjarne Stroustrup

- Datentypen
 - Fundamentale Datentypen
 - `char` ein Zeichen
 - `int` ganzzahliger Wert (Integer)
 - `float` Gleitkommazahl einfacher Genauigkeit
 - `double` Gleitkommazahl doppelter Genauigkeit
 - Mit den entsprechenden Attributen: `short`, `long`, `unsigned`, `signed`
 - Spezielle Datentypen
 - `void` z.B. Funktionen ohne Rückgabewert
 - `enum` Aufzählungstypen
 - Boolesche Werte (`bool`)
 - `true` entspricht 1
 - `false` entspricht 0
 - Zeiger
 - Operator `&` liefert die Adresse eines Datums
 - Operator `*` dereferenziert einen Zeiger, d.h. liefert den Wert auf den der Zeiger zeigt

- Zusammengesetzte Datentypen

- Arrays

- Menge mehrerer Elemente gleichen Typs, die hintereinander angeordnet sind
 - Z.B. `int werte[80];` `/* Array von 80 int's */`
 - Zugriff auf ein Element über den Operator `[]`
 - Z.B. `werte[0] = 77;`

- Zeichenketten

- Arrays von Zeichen mit dem Zeichen `, \0 \` als Endekennung.

- Strukturen (`struct`)

- Ein Datenverbund aus mehreren Elementen verschiedenen Typs
 - Zugriff über die Operatoren `„. “` bzw. `„->“`

- Union (`union`)

- Variable die zu verschiedenen Zeitpunkten mehrere Objekte verschiedenen Typs und Größe enthält
 - Zugriff über die Operatoren `„. “` bzw. `„->“`

C als Voraussetzung (III)

- Kontrollstrukturen
 - Fallunterscheidungen
 - `if` für die einfache Fallunterscheidung
 - `switch` für die mehrfache Fallunterscheidung
 - Schleifen
 - `while`
Prüfung der Schleifenbedingung vor jedem Durchlauf
 - `do-while`
Prüfung der Schleifenbedingung nach jedem Durchlauf
 - `for`
Schleifenkopf mit Initialisierung, Bedingung und Reinitialisierung
 - Blöcke
 - Zusammenfassung mehrerer Anweisungen eingeschlossen in `{ }`
- Funktionen
 - Prototype Deklaration und Definition
 - Funktion `main()` als „Hauptprogramm“

```
if (i > 5)
{
    ...
}
else
{
    ...
}
```

```
switch (i)
{
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default:
        ...
}
```

```
while (i > 7)
{
    ...
}
```

```
for (i = 0; i < 7; i++)
{
    ...
}
```

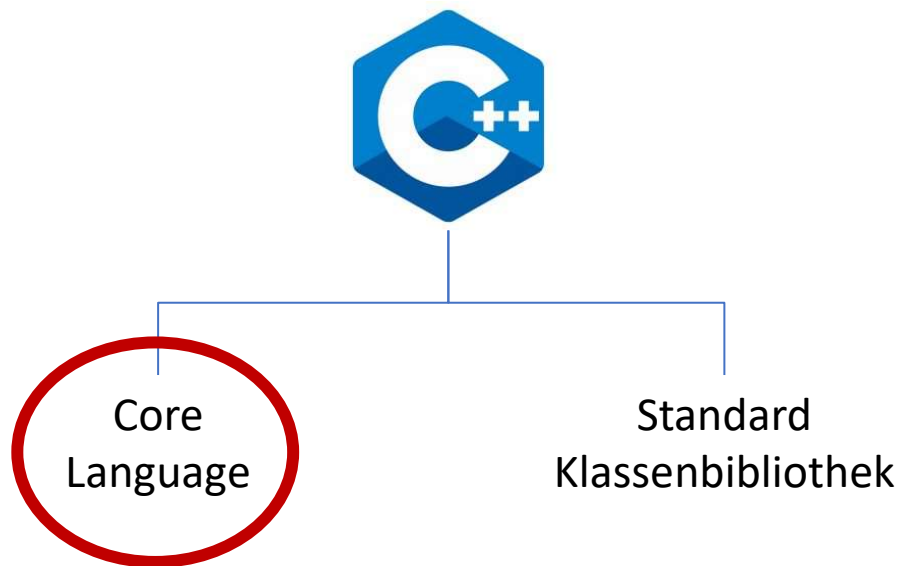
```
do
{
    ...
} while (i > 7);
```

C als Voraussetzung (IV)

- Standardfunktionen bzw. Bibliotheken
 - Stehen weiter uneingeschränkt zur Verfügung
 - `<string.h>`
 - Z.B. `strcpy()`, `strlen()`, `strcmp()`, `strcat()`
 - `<stdlib.h>`
 - Z.B. `exit()`
 - `<stdio.h>`
 - Z.B. `printf()`, `scanf()`, `fopen()`, `fclose()`, `stdin`, `stdout`, `stderr`
 - Teilweise gibt es allerdings C++ Alternativen
 - z.B. `iostream` für die `stdio.h`
- Groß- und Kleinschreibung beachten
 - C und C++ sind Case sensitive!!!



- Klassen & Objekte inklusive Datenkapselung
- Überladen von Funktionen / Operatoren
- Vererbung / Polymorphie
- Exception Handling
- Templates
- Bibliotheken



Fokus der Vorlesung

C++ Version 11, 14, 17, 20, ...

- Alle 3 Jahre wird eine neue Version als Standard herausgegeben
- Betrifft Core Language als auch die Standardklassenbibliothek
- Erweiterungen aber auch „Abmeldungen“
- Unterstützung durch Compiler „hinkt“ oft hinterher ...

Vorlesung verwendet Sprachelemente die seit der ersten Standardisierung stabil sind.

- C++ basiert auf der Programmiersprache C als Grundlage.
 - Die Sprache C ist damit auch vollständig in C++ enthalten.
 - Zusätzlich beinhaltet C++ Konstrukte für die objektorientierte Programmierung (und andere Erweiterungen die dem Programmierer das Leben einfacher machen können).
- C++ besteht aus der eigentlichen **Kernsprache** (Core Language) und der **Standardklassenbibliothek**.
 - Die Standardklassenbibliothek wurde mit den Mitteln der Kernsprache entwickelt.
- Alle 3 Jahre wird eine neue Sprachversion (Kernsprache und Standardklassenbibliothek) veröffentlicht.
 - Jede neue Version beinhaltet dabei Erweiterungen als auch Abmeldungen.
 - Aktuell ist die Version 20.



Klassen & Objekte



```
if operation == "Mirror_Y":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
elif operation == "Mirror_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True
```

```
#selection at the end -add back the deselected mirror modifier object  
mirror_ob.select= 1  
modifier_ob.select=1  
bpy.context.scene.objects.active = modifier_ob  
print("Selected" + str(modifier_ob)) # modifier ob is the active ob  
#mirror_ob.select = 0
```

```
#now = bpy.context.selected_objects[0]  
#bpy.data.objects[name].select = 1
```

```
#print(modifier_ob.name) # modifier_ob is the active ob  
#print(modifier_ob.name) # modifier_ob is the active ob
```

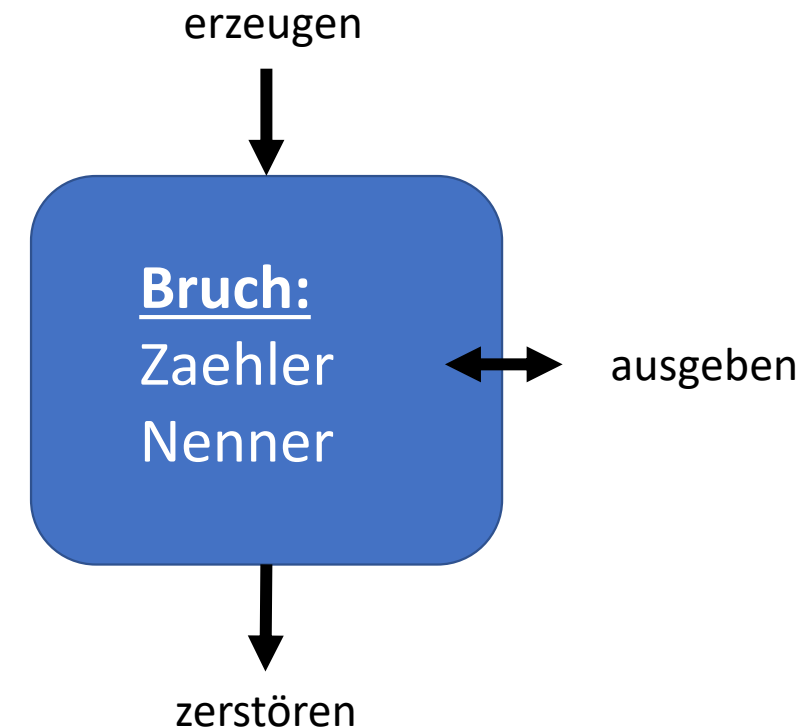
Ein Beispiel als Anschauungsobjekt

Prinzipielle Fragen

- Was für eine Klasse soll implementiert werden?
- Was soll man mit Objekten der Klasse machen können? Funktion/Schnittstellen?
- Wie ist der interne Aufbau dieser Objekte

Klasse **Bruch**

- Aufbau
 - Zähler
 - Nenner
- Funktionen/Schnittstellen
 - Erzeugen/Initialisieren eines Bruchs
 - Ausgeben eines Bruchs
 - Zerstören eines Bruchs



Klassendeklaration in C++

- Keyword: class
 - Gefolgt vom Namen der Klasse
- Zugriffsdeklarationen
 - private/public
- Komponenten (members)
 - Attribute (zaehler, nenner)
 - Methoden / Elementfunktionen / Memberfunktion

```
class Bruch
{
    /* kein Zugriff von aussen */
private:
    int zaehler;
    int nenner;

    /* oeffentliche Schnittstelle */
public:
    Bruch();
    Bruch(int);
    Bruch(int, int);
    ~Bruch();

    void print();
};
```

Zugriffsdeklarationen

- Grundlegendes Prinzip der OOP: Datenkapselung
 - Zugriff auf Objekte nur über definierte Operationen (Funktionen)
 - Vermeidung von Inkonsistenzen
- Mögliche Inkonsistenzen - Beispiel Bruch
 - Nenner darf nicht 0 sein
 - Freier Zugriff wäre potentiell fehleranfällig
- Zugriffsdeklarationen
 - `public` von außen sichtbar („Schnittstelle“)
 - `private` nur von Methoden der Klasse sichtbar
- Zugriffsdeklarationen können beliebig oft innerhalb einer Klasse vorkommen
 - Ohne vorangestellte Zugriffsdeklaration gilt `private` (!)
- Alle Deklarationsarten sowohl für Attribute als auch Methoden möglich
 - Attribute können auch `public` sein
 - Methoden können auch `private` sein

```
class Bruch
{
    private:
        int zaehler;
        int nenner;

    public:
        ...

    private:
        ...

    public:
        ...

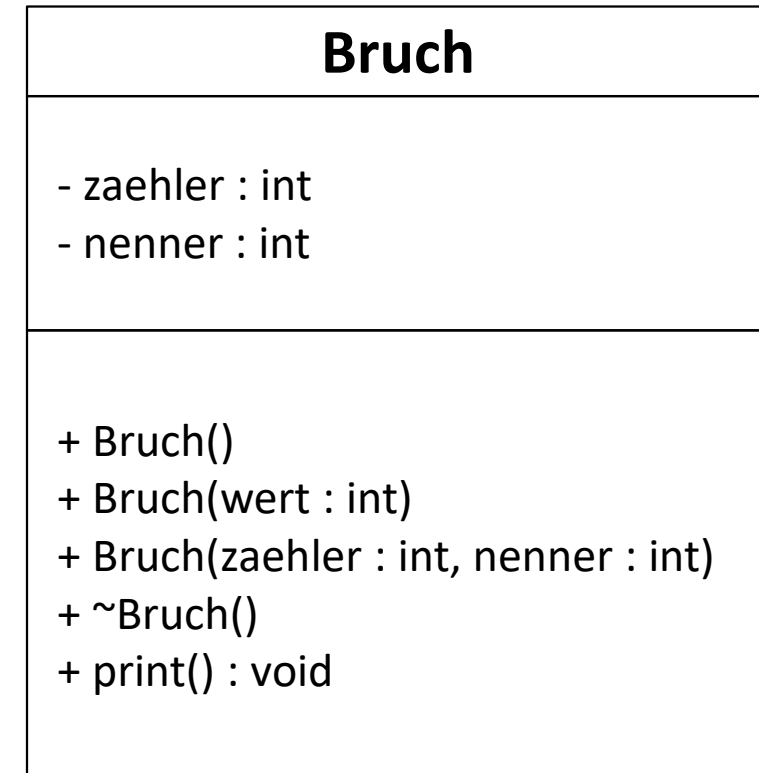
};
```

Darstellung in UML

```
class Bruch
{
    /* kein Zugriff von aussen */
private:
    int zaehler;
    int nenner;

    /* oeffentliche Schnittstelle */
public:
    Bruch();
    Bruch(int);
    Bruch(int, int);
    ~Bruch();

    void print();
};
```



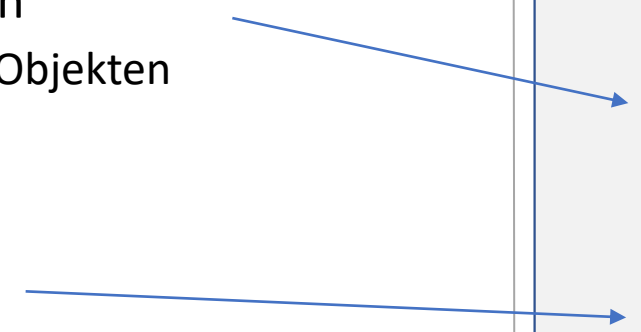
2 „*unterschiedliche*“ Arten

- Konstruktoren / Destruktoren
 - Erzeugen & Zerstören von Objekten
- „*normale*“ Methoden
 - Schnittstellen der Klasse
 - Unterstützte Operationen

```
class Bruch
{
    ...

    Bruch();
    Bruch(int);
    Bruch(int, int);
    ~Bruch();

    void print();
};
```



Konstrukturen

Legen fest, wie ein Objekt der Klasse erzeugt wird

- Z.B. zur Initialisierung des Objekts

Haben den Namen der Klasse

- Klasse: Bruch
- Konstruktor: Bruch

Mehrere Konstrukturen möglich

- Unterschiedliche Arten der Initialisierung

Default Konstruktor *<Klasse>()*

- Ohne Argumente
- Falls kein anderer Konstruktor spezifiziert, vom Compiler hinzugefügt (aber ohne inhaltliche Funktion)

Konstrukturen haben keinen Typ

- d.h. keinen Rückgabewert

```
class Bruch
{
    ...

    /* initialisiere mit 0 */
    Bruch();

    /* initialisiert mit Ganzzahl */
    Bruch(int);

    /* initialisiert mit Zaehler/Nenner */
    Bruch(int, int);
    ...
};
```

Erzeugen von Objekte (I)

Klassen werden wie Datentypen behandelt:

- Wie `int`, `char`, `struct`, ...

Im nicht OOP Sprachgebrauch (z.B. C):

- Es wird eine Variable vom Type `Bruch` angelegt.

Im OOP Sprachgebrauch:

- Es wird eine Instanz der Klasse `Bruch` angelegt.

Zeiger auf ein Objekt der Klasse sind auch möglich

- Wie bei anderen Datentypen auch

```
...  
  
void calculate()  
{  
    Bruch a;  
    Bruch b1 = 1;  
    Bruch b2(1);  
    Bruch c(1,3);  
    Bruch* bPtr;  
  
    bPtr = &a;  
    ...  
}  
  
...
```

Erzeugen von Objekte (II)

- Zuerst wird der Speicherplatz angelegt (Zustand ist aber nicht definiert)
- Anschließend erfolgt (automatisch) der Aufruf des passenden Konstruktors
 - Ohne Initialisierung (Bsp. a):
 - Aufruf des Default Konstruktors (ohne Parameter)
 - Mit Initialisierung (Bsp. b1, b2, c)
 - Aufruf des zu den Initialisierungsparametern passenden Konstruktors.
 - `Bruch b1 = 1; ⇒ Bruch(int)`
 - `Bruch b2(1); ⇒ Bruch(int)`
 - `Bruch c(1,3); ⇒ Bruch(int,int)`
 - Existiert kein passender Konstruktor meldet der Compiler einen Fehler

Array von Objekten (z.B. `Bruch werte[80]`)

- Konstruktor wird für jedes Element des Arrays aufgerufen.

Im Fall einer global deklarierten Instanz einer Klasse:

- Ausführen des Konstruktors vor Start der `main()` Funktion!!

```
...  
  
void calculate()  
{  
    Bruch a;  
    Bruch b1 = 1;  
    Bruch b2(1);  
    Bruch c(1,3);  
  
    ...  
}  
  
...
```

Destruktor

Legt Aktionen fest, die noch vor dem Zerstören des Objekts ausgelöst werden sollen

- Aufräumarbeiten (z.B. Speicherfreigabe)

Hat den Namen der Klasse mit vorangestellter ~

- Klasse: Bruch
- Destruktor: ~Bruch

Nur ein Destruktor pro Klasse

- Wird nicht automatisch erzeugt

Destrukturen sind typlos und haben keine Parameter

```
class Bruch
{
    ...

    /* räume auf vor Zerstörung */
    ~Bruch();

    ...
};
```

Zerstören von Objekte

Objekte werden automatisch zerstört, wenn der Gültigkeitsbereich der Variable verlassen wird.


- z.B. beim Ende der Funktion in der die Variable definiert wurde

Beim Zerstören wird automatisch der Destruktor aufgerufen.

Im Fall eines global deklarierten Instanz einer Klasse:

- Ausführen des Destruktors nach Beendigung der `main()` Funktion!!

```
...  
  
void calculate()  
{  
    Bruch a;  
  
    ...  
}  
  
...
```



Zugriff auf Methoden / Attribute

Methodenname / Attributnamen könnte in verschiedenen Klassen verwendet werden.

Daher klare Zuordnung notwendig.

`<objekt>.<methode>`

`<objekt>.<attribut>`

„Schicke dem Objekt die Nachricht `print`“

Zugriff von außen nur auf öffentliche Methoden und Attribute möglich (Deklaration *public*).

Zugriff über Zeiger analog `struct/union`

```
void calculate()
{
    Bruch a;
    Bruch b = 1;
    Bruch c(1,3);
    Bruch *bPtr = &a;

    a.print();
    b.print();
    c.print();

    bPtr->print();

    ...
}
```


Implementieren der Methoden (I)

Zuordnung der Methode zur Klasse über den
Bereichsoperator

`<klass>::<methode>`

Methoden haben (auch ohne Bereichsoperator) vollen Zugriff auf die Attribute der eigenen Klasse (`public` und `private`).

Konstruktoren und Destruktor haben keinen Typ – damit auch keinen Rückgabewert.

Normale Methoden haben einen Typ und potentiell einen Rückgabewert.

```
/* Default Konstruktor */
Bruch::Bruch()
{
    /* initialisiere mit 0 */
    zaehler = 0;
    nenner = 1;
}

Bruch::Bruch(int i)
{
    /* initialisiere mit i 1-tel */
    zaehler = i;
    nenner = 1;
}
```

Implementieren der Methoden (II)

```
Bruch::Bruch(int z, int n)
{
    if (n == 0)
    {
        fprintf(stderr,
                "Fehler: Nenner 0\n");
        exit(1);
    }
    zaehler = z;
    nenner = n;
}
```

```
Bruch::~~Bruch()
{
    /* keine Aktion notwendig */
}
```

```
void Bruch::print()
{
    printf("Zaehler: %d\nNenner: %d\n",
           zaehler,
           nenner);
}
```

Bezugsrahmen von Variablen

Eine Variable ist innerhalb eines Bezugsrahmens deklariert

- Global, Klasse, Funktion, Block

Die Verwendung des gleichen Bezeichners an mehreren Stellen kann zu einer Überdeckung führen

Reihenfolge (zuerst - zuletzt)

- Lokale Variable im aktuellen Block
- Lokale Variable in der Funktion
- Attribute der Klasse (für Methoden)
- Global Variable

Bereichsoperator kann helfen

Empfehlung

- Sinnvolle Bezeichner einsetzen
- Kurze Bezeichner nur innerhalb Blöcken und Funktionen

```
static int a = 3;

class gueltig
{
private:
    int a;
public:
    gueltig(int);
    void print();
};

void gueltig::print()
{
    int a = 1;
    printf("%d\n", a);
}
```

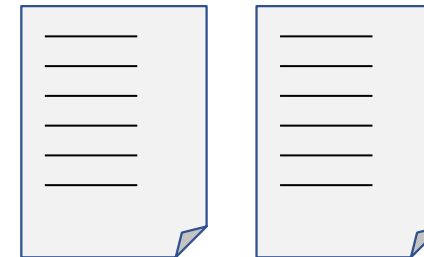
Diagramm zur Variablenbeziehung:

- Globale Variable:** Zeigt auf die globale Variable `static int a = 3;`.
- Attribut der Klasse:** Zeigt auf das Attribut `int a;` innerhalb der Klasse `gueltig`.
- Lokale Variable:** Zeigt auf die lokale Variable `int a = 1;` innerhalb der Methode `print()`.

Typische Quelldatei Struktur

- Dateiendung .cpp (statt .c)
- Eine .cpp für das Hauptprogramm
 - d.h. `main()` Funktion
- Pro Klasse 2 Dateien
 - .h Datei
 - Schnittstellenbeschreibung (z.B. Klassendeklaration)
 - .cpp Datei
 - Implementierung der Klasse
 - Dateiname = Klassenname
- Wie auch bei C weitere Quelldateien möglich

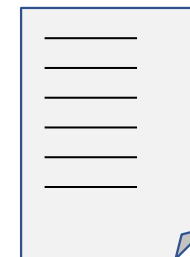
<class a>.cpp <class a>.h



<class ...>.cpp <class ...>.h



<main>.cpp



<other>.cpp <other>.h



- **Klassen** werden durch Strukturen mit dem Schlüsselwort `class` deklariert.
 - Komponenten besitzen **Zugriffsrechte** die durch die Schlüsselworte `private` und `public` vergeben werden.
- Komponenten der Klassen können auch Funktionen (sogenannte **Methoden**) sein.
 - Sie bilden typischerweise die öffentliche Schnittstelle zu den Komponenten.
 - Methoden haben Zugriff auf alle Komponenten ihrer Klasse.
- **Konstruktoren** und der **Destruktor** sind spezielle Methoden
 - Konstruktoren werden beim Erzeugen von Objekten einer Klasse aufgerufen und dienen dazu, diese zu initialisieren.
 - Destrukturen werden beim Zerstören des Objekts aufgerufen.
 - Konstruktoren und Destruktor besitzen keinen Rückgabetyt.
- Der **Bereichsoperator** `::` ordnet einem Identifier den Gültigkeitsbereich einer bestimmten Klasse zu.

