

Einführung in C++ mit Microsoft VS

Stefanie Kettner

17 April 2007

chair for computer aided medical procedures

department of computer science | technische universität münchen

Gliederung

- Allgemeines zu C++ und Unterschiede zu Java
 - Header-Files
 - Zeiger/Strukturen
 - Namespaces
 - Umgang mit Bibliotheken
 - Programmierstil
- Einführung in Microsoft VS
 - Projekte und Solutions
 - Debugger
 - Tipps und Tricks

C++ vs. Java

Rückblick: Klassen und Objekte

- **Klasse** ist ein Objekttyp --- Objekt ist eine Instanz einer Klasse
- **Objekte** bestehen aus Daten (Membervariablen) und zugehörigen Funktionen (Memberfunktionen)
- statische Variablen von einer Klasse gibt es nur *einmal* pro Prozess
 - eine Änderung einer statischen Variable wirkt sich auf *alle* Instanzen aus
- **statische Funktion:** Aufruf ohne Instanz → dürfen aber nicht auf Member zugreifen
- nur **virtuelle Funktionen** in vererbten Klassen können überschrieben werden und werden zur Laufzeit bestimmt
- **abstrakte Klassen:** nicht alle virtuellen Funktionen sind implementiert → keine Objekterzeugung möglich

Strukturen und Aufzählungen

- **Array** : Zusammenfassung von Elementen desselben Typs
- **Struct** : Zusammenfassung von Elementen bel. Typs

```
struct TelefonbuchEintrag {  
    string Nachname;  
    string Vorname;  
    int    Vorwahl;  
    int    Rufnummer;  
};
```

```
TelefonbuchEintrag.Nachname = "Kettner";  
TelefonbuchEintrag.Vorname  = "Stefanie";  
TelefonbuchEintrag.Vorwahl  = 089;  
TelefonbuchEintrag.Rufnummer = 28919405;
```

- **Enum** : Menge spezifischer Werte (wird benutzt wie int)

```
enum Bier {  
    Augustiner,  
    Loewenbraeu,  
    Becks  
};
```

```
void f(Bier bierType) {  
    if (bierType == Augustiner)  
        drink();  
}
```

Die Header-Datei

- sämtliche Definitionen und Deklarationen, die dann in der .cpp-Datei implementiert werden
- sämtliche Präprozessoranweisungen (z.B. #include) sollten nach Möglichkeit NUR in die Header-Datei geschrieben werden

1. Klassendeklaration

```
class MyClass { ... };
```

2. Typdefinition

```
struct Position { int x, y };
```

3. Aufzählungen

```
enum Ampel { rot, gelb, grün };
```

4. Funktionsdeklaration

```
int rechteckFlaeche(int w, int h);
```

5. Konstantendefinition

```
const float pi = 3.141593;
```

6. Datendeklaration

```
int a;
```

7. Präprozessoranweisungen

```
#include <iostream>  
#define VERSION 12  
#ifdef __cplusplus
```

Die Header-Datei

myClass.h

```
class MyClass : MyParentClass
{ // die Klasse MyClass erbt von MyParentClass
public:
    MyClass(); //Standardkonstruktor
    MyClass(std::string text); //zweiter Konstruktor
    virtual ~MyClass; //Destruktor

    virtual int func()=0; //eine rein virtuelle (=abstrakte) Funktion
    static double func(); //eine statische Funktion

    static int m_someNumber; //eine statische Membervariable

protected:
    virtual int fun(); //eine virtuelle Funktion

private:
    void fu(); //eine Funktion
    std::string m_someString; //eine Membervariable
}
```

Die Implementierung

```
#include "myClass.h"

int MyClass::m_someNumber(5);

MyClass::MyClass(){
    m_someString = "Eine Intialisierung vom Text";
} //Standardkonstruktor

MyClass::MyClass(std::string text){
    m_someString = text;
} //zweiter Konstruktor

MyClass::~MyClass(){} //Destruktor

void MyClass::fu(){} //eine Funktion
int MyClass::fun(){return 4;} //eine virtuelle Funktion
double MyClass::funct(){return 2;} //eine statische Funktion
```

myClass.cpp

Zeiger und Referenz

Ein Zeiger (MyObject*)

- zeigt auf einen Speicherbereich
- Es gibt Nullzeiger (Nullpointer)

```
MyObject object;  
MyObject* objectPointer;  
  
MyObject& ref = object;
```

Eine Referenz (MyObject&)

- ist wie ein neuer Name für ein bestehendes Objekt
- Es gibt keine Nullreferenz

JAVA kennt nur die Mischung aus beiden

Umwandlung

- Die gleichen Zeichen werden bei Umwandlungen benutzt

```
MyObject object;  
MyObject* objectPointer;  
  
objectPointer = &object;  
object = *objectPointer;
```

- & vor einem Objekt bedeutet
“Ich hätte gern den Speicherort vom Objekt”
- * vor einem Zeiger (bzw. Speicherort) bedeutet
„Ich hätte gern die Referenz (bzw. das Objekt) zum Zeiger“

Wozu das ganze? Geschwindigkeit!

```
Telefonbuch t;  
string suchbegriff;  
Eintrag e;  
e = suchfunktion(suchbegriff, t);
```

- Funktionsparameter und Rückgabewerte können Geschwindigkeit beeinträchtigen

```
Eintrag suchfunktion(string s, Telefonbuch t)  
{ // kopiert beim Aufruf das ganze Telefonbuch  
  ...  
  return e;  
}
```

- Referenzen sind effizienter aber mit Vorsicht zu genießen

```
Eintrag suchfunktion(string& s, Telefonbuch& t)  
{ // kopiert nur 4 Byte  
  ...  
  return e;  
}
```

```
Eintrag& suchfunktion(string& s, Telefonbuch& t)  
{ // wenn man den Eintrag ändert, ändert sich  
  // auch der im Telefonbuch!  
  ...  
  return e;  
}
```

Wozu das ganze? Geschwindigkeit!

- Effizientes Programmieren mittels Zeigerarithmetik

```
// ein Byte Array (10MByte) um den Wert 5 erhöhen
char* array = new char[10000000];
for (int i=0; i < 10000000; i++){
    array[i] += 5;
}
delete[] array;
```

- Folgende Implementierung ist schneller


```
char* array = new char[10000000]; //zeigt auf erstes Element
char* endOfArray = array+10000000; //zeigt auf letztes+1 Element
for (char* i=array; i < endOfArray; i++){
    (*i) += 5; //Wert des Elements, auf das Zeiger i zeigt
}
delete[] array;
```

Keller und Halde (Stack und Heap)

- Es gibt zwei Möglichkeiten Speicher für seine Objekte zu bekommen:

```
MyObject mo1; // Stack  
MyObject* mo2 = new MyObject(); // Heap
```

- **Stack:** Speicher wird wieder überschrieben, nachdem die Variable aus dem lokalen Namensraum verschwindet
- **Heap:** Speicher wird erst wieder freigegeben, wenn man `delete` aufruft
- Bei JAVA kommt alles auf den Heap; der GarbageCollector gibt es dann wieder frei



Fehlerquelle
Nr. 1 !

Tipps zu Zeigern (Pointer) und Referenzen

- Pointer → Vererbungsverhältnisse zur Laufzeit ausnutzen
- Pointer → Variable kann Nullpointer sein
- Referenz → Objekt muss sicher vorhanden sein

- zu `new` gehört in JEDEM Fall ein `delete`

```
MyObject* mo2 = new MyObject(42);  
:  
delete mo2;
```

- Man sollte sich angewöhnen, alle Pointer mit 0 zu initialisieren und vor `delete` auf 0 zu überprüfen
- Zu `new MyObject[42]` gehört `delete[]`

const Deklaration

- Funktion, bei der die Parameterwerte nicht verändert werden

```
Eintrag& meineFunktion(const Telefonbuch& t, const string& s);
```

- Funktion, bei der keine Membervariablen (auch nicht indirekt) geändert werden

```
Eintrag& meineFunktion(meinObject& mo1, meinObject& mo2) const;
```

- Funktion, bei der die Rückgabewerte später nicht mehr geändert werden dürfen

```
const Eintrag& meineFunktion(meinObject& mo1, meinObject& mo2);
```

Tipps zu const

- Compiler prüft den Code zu `const` Deklarationen und zeigt gegebenenfalls einen Fehler (keine Warnung!)
- Man muss nicht mit `const` programmieren, aber es hilft
- `const` immer von rechts nach links lesen!
 - `const MyObject*` ist ein Pointer auf ein konstantes Objekt
 - `MyObject const*` ist ein konstanter Pointer auf ein Objekt
 - `const MyObject const*` ist ein konstanter Pointer auf ein konstantes Objekt

Namensraum (namespace)

- erleichter bei größeren Projekten die eindeutige Namensvergabe

```
namespace myNamespace{  
    class MyClass{  
        MyClass();  
        void myNonsenseFunction(){ return; }  
    };  
}
```

myClass.h

```
include "myClass.h"  
int main(){  
    myNamespace::MyClass someObject;  
    someobject = new myNamespace::MyClass;  
    return 0;  
}
```

main.cpp

```
include "myClass.h"  
using namespace myNamespace;  
int main(){  
    MyClass someObject;  
    someobject = new MyClass;  
    return 0;  
}
```

main.cpp

Die Standardbibliothek STL

- Viele elementare Methoden sind darin vorimplementiert
- die wichtigsten header sind `cmath`, `iostream`, `string`, `iterator`, ...
- ausführliche Beschreibung in [B. Stroustrup]

```
include <iostream>
include <string>
int main(){
    std::string s = "Was ist die Losung";
    std::cout << s << std::endl;
    std::cin >> t;
    if( t.compare("per aspera ad astra")==0 ) {
        std::cout << "Richtig!" << std::endl;
    }
    return 0;
}
```

Compiler

- Sammlung von Programmen [Compiler Collection], die man zur Programmerstellung benötigt
 1. Präprozessor
Textersetzungen gemäß Präprozessoranweisungen
 2. Übersetzer [Compiler]
Übersetzt Programmfragmente in C++ zu Programmfragmenten in Binärcode
 3. Verknüpfer [Linker]
Verknüpft Binärprogrammteile zu einem Programm

Präprozessor

- Macht Textersetzungen *vor* dem Übersetzen
- alle Anweisungen für den Präprozessor fangen mit # an
 - `#include "file.h"` fügt die Header-Datei file.h ein
 - `#define BLUBB 1` ersetzt überall im Programmtext BLUBB mit 1
 - `#ifdef BLUBB`
Code1
`#else`
Code2
`#endif`
fügt im Programmtext Code1 ein falls BLUBB definiert wurde, ansonsten Code2

Präprozessor

- Schutz davor, dass man den Header mehrmals aus Versehen aufruft:

```
#ifndef MEINEKLASSE_H
#define MEINEKLASSE_H
#include ... MeineKlasse.h
class MeineKlasse {...};
#endif
```

- Variablen (eigentlich Ersetzungsworte) zur Übersichtlichkeit immer mit Großbuchstaben benennen!
- Wenn es geht, so wenig wie möglich mit dem Präprozessor arbeiten!

Bibliotheken [Libraries]

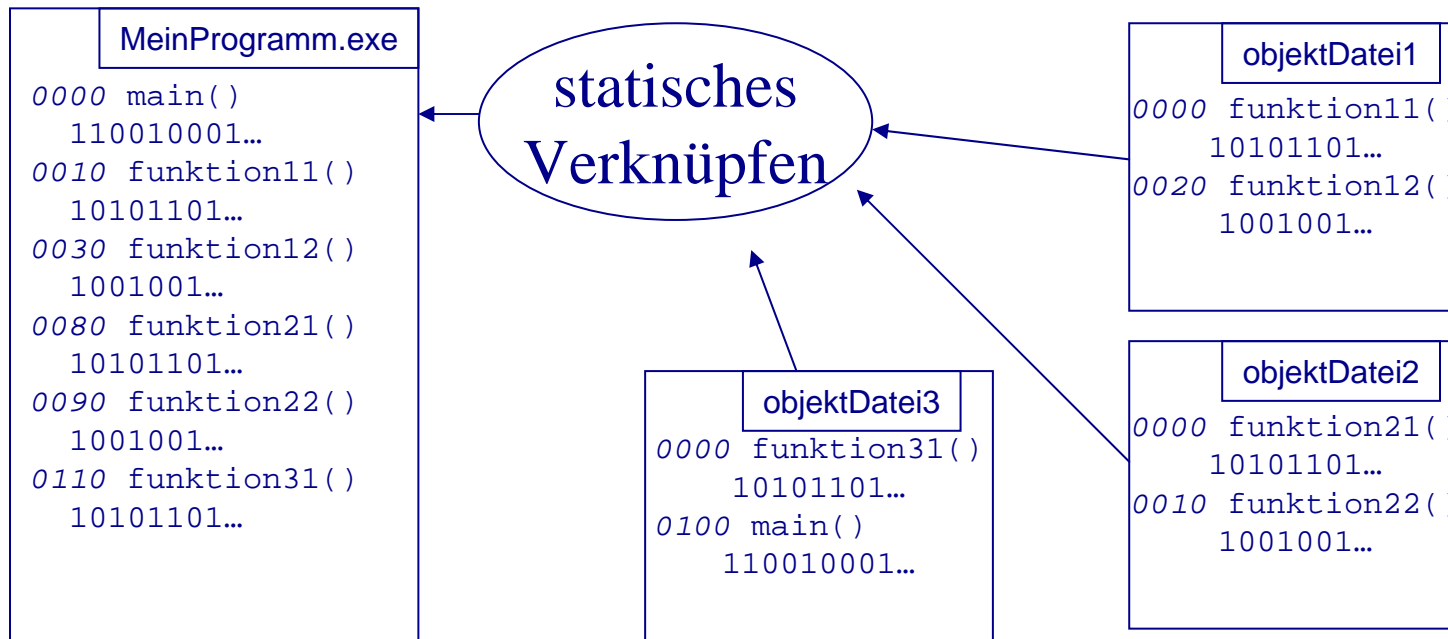
- Bibliotheken (*.lib) sind vorkompilierte Binärdateien ohne Einstiegspunkt [Entrypoint]
- Zeitvorteil, weil Programmteile, die sehr oft benutzt werden, nicht immer wieder neu kompiliert werden müssen.
- Rechtlich einwandfrei, weil Programmteile weitergegeben werden ohne den Programmtext zeigen zu müssen.
- Programmiertechnischer Vorteil, weil weitergegebene Programmteile nicht geändert werden können.
- Platz- und Geschwindigkeitsvorteil, weil man beliebte Programmteile nur einmal auf der Festplatte (oder im Speicher) vorrätig haben muss.

Einstiegspunkt?!

- Der Einstiegspunkt [Entrypoint] eines Programms ist der Teil, an dem das Programm starten soll. Unter C++ ist es die globale Funktion `int main()`, die es selbstverständlich nur einmal geben darf
- Ein Programm ohne Einstiegspunkt ist allein nicht lauffähig

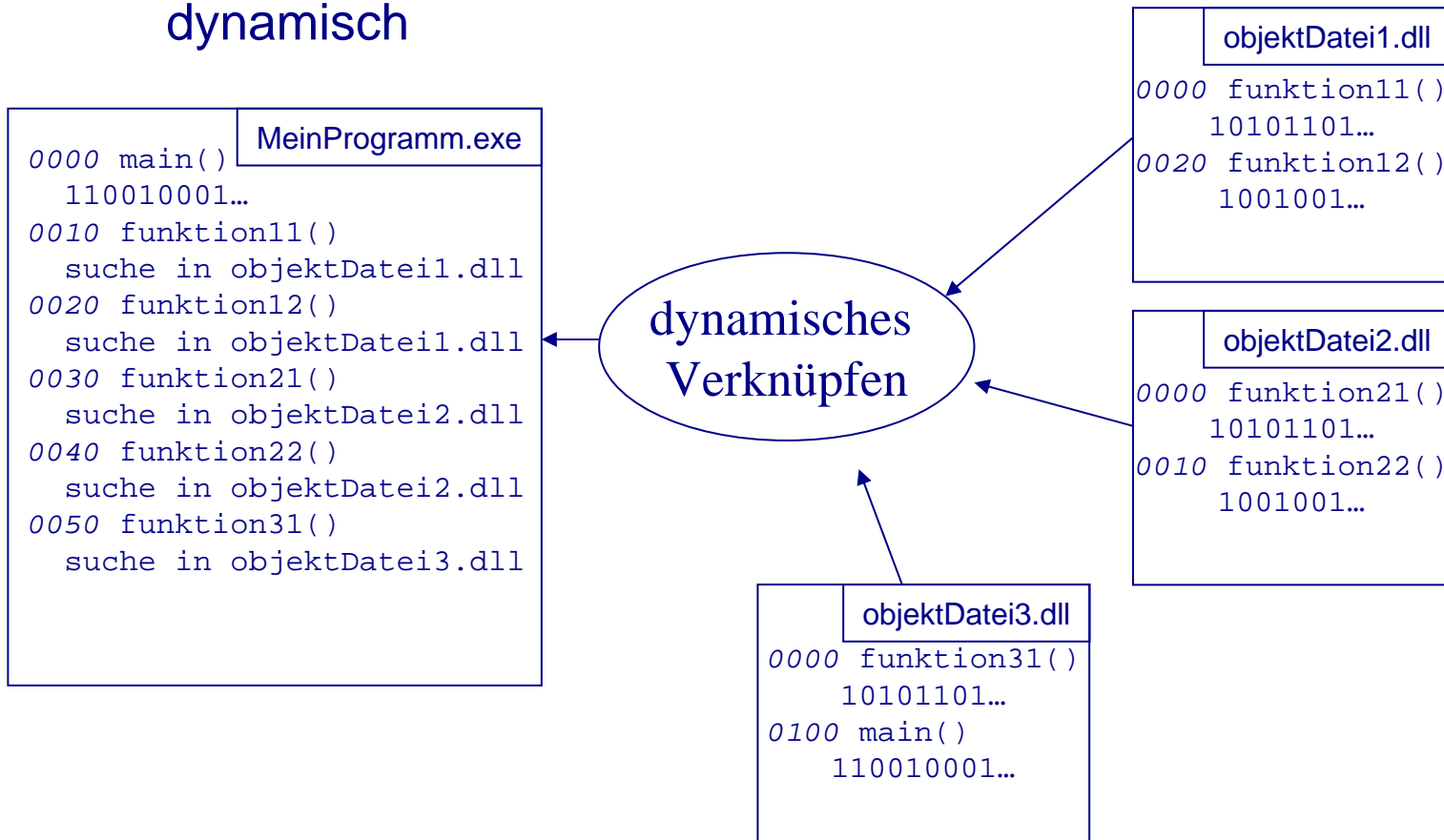
Verknüpfen (link)

- Das eigentliche Programmerstellen übernimmt der Linker
- Er kopiert den Code aus den Bibliotheken und ersetzt im Code die symbolischen Funktionen mit konkreten Adressen



Verknüpfen (link)

- Die zweite Art der Verknüpfung funktioniert zur Laufzeit, d.h. dynamisch



Gegenüberstellung

statisches Verknüpfen

- einfachste (auch älteste) Methode
- ein große Datei
- portable Verknüpfungsmethode
- neue Version der Bibliothek → ganze Datei muss neu generiert werden

dynamisches Verknüpfen

- komplizierter Mechanismus, der auf unterschiedlichen Betriebssystemen unterschiedlich angesprochen wird
- Realisierung von Plugins möglich
- Weniger Speicherbedarf
- neue Version der Bibliothek → nur Bibliothek muss neu generiert werden

Tipps zu Compiler-/Linkerfehlern

- meist wurde in den angegebenen Objektdateien ein Symbol (Funktion, Variable, Klasse, ...) nicht gefunden
- Der Grund dafür ist meistens
 - ein Schreibfehler
 - falscher oder fehlender Namensraum
 - `virtual / static` vergessen
- Der Compiler prüft **n**ie nach, ob eine deklarierte Funktion (im Header `xx.h`) tatsächlich (in der Implementierung `xx.cpp`) implementiert wurde
- Der Compiler beschwert sich aber **immer**, wenn eine nicht deklarierte Funktion implementiert wird
- Manchmal hilft es, **alles** (inklusive Bibliothek) komplett neu zu bauen

TODO-Liste für fremde Bibliotheken

- im Programmtext den Header einbinden, in dem die gewünschten Objekte oder Funktionen bereitgestellt werden
- dem Linker den/die Dateinamen der Bibliotheken mitteilen, damit diese mit dem Programm verknüpft werden können
- den Pfad des Headers **und** der Bibliothek angeben

Coding Guidelines - Funktionen

- Eine Funktion sollte nicht mehr als eine Seite beinhalten
- Jede dritte Zeile sollte ein Kommentar sein
- Alle öffentlichen und geschützten Funktionen und Variablen sollten Kommentare im Header haben
- Ein Aufruf sollte nicht viele andere Aufrufe schachteln
- Variablendeklaration sollte in der Nähe des ersten Aufrufs sein
- Destruktoren sollten immer virtuell sein (`virtual ~MyObject` im Header)

Coding Guidelines - Namen

- Sollten nicht den Namen des Autors beinhalten
- Der Sinn sollte erkennbar sein (computeArea, width, result)
- Keine Ähnlich zu anderen Namen (myReference, myReferene)
- Englisch (oder die offizielle Sprache des Projektes)
- Membervariablen wird `m_` vorangestellt (`m_width`)
- In allen Variablen und Funktionen sind die Anfangsbuchstaben klein, in weiteren Wortteilen groß
- In Klassennamen sind die Anfangsbuchstaben groß
- enums in Großbuchstaben

Coding Guidelines - Datenfluss

- Const benutzen
- Es gibt keinen vernünftigen Grund für globale Variablen, Funktionen oder anderes
- Es gibt keinen vernünftigen Grund für #define
- Möglichst wenig Daten fließen lassen, in dem sich der interne Zustand des Objektes ändert (also in einer Funktion Sachen in eine Membervariable ablegen, die man auch mit einer Referenz übergeben kann)
- Man muss nicht immer alles mit Vererbungshierarchien lösen – nur wenn es die Aussage trifft

Coding Guidelines - Schneller Code

- Stilvoll programmieren bedeutet nicht (effektiv) langsamerer Code
- 80/20 Regel beachten: 20% des Aufwandes sollten 80% des Ergebnisses ausmachen
 - Erst schauen welche Zeile am meisten Zeit verbraucht, dann diese optimieren
 - Ansonsten erstmal standardmäßig Geschwindigkeit gegen Übersichtlichkeit und Einfachheit opfern
- Wenn man das durchzieht, hat man immer schnellen Code, denn man beschäftigt sich weniger mit Debuggen und Optimierung von Stellen, die es nicht Wert sind

Coding Guidelines - Fehlersuche

- Din A4 Blatt mit seinen Lieblingsfehlern anlegen
- Compiler-Warnungen haben ihren Sinn!
- Linkerfehler habe nur 3 Ursachen
 - Die .lib Datei wurde nicht im Projekt nicht (korrekt) angegeben
 - Der Pfad für die .lib Datei wurde nicht (korrekt) angegeben
 - Es gibt die Funktion im Header, aber nicht in der cpp Datei (mit der korrekten Signatur)

Coding Guidelines - Bibliotheken

- Man muss nicht das Rad 2x erfinden
- STL sollte man kennen und auch verwenden
 - Vector (list, map) statt Arrays
 - Suchalgorithmen gibt es schon
 - `string` erlaubt meist bessere Handhabung als `char*`
- Vieles gibt es schon in vernünftiger Qualität mit einer freundlichen Lizenz

Microsoft VS

Integrierter Debugger

- Kompilieren im Debugmodus:
 - Der Compiler fügt zusätzliche Informationen dem Code hinzu, die das Programm semantisch nicht verändern
 - Es finden keinen zusätzlichen Optimierungen mehr statt

- Breakpoints im Quellcode:
 - Man kann das Programm Schritt für Schritt im Programmtext ausführen (obwohl das Programm nach wie vor im Binärcode läuft!)

Haltepunkte (Breakpoints)

- Das Programm hält immer am Haltepunkt, wenn es im Debugmodus kompiliert wurde UND im Debugmodus gestartet wurde (F5)
- Unter „Haltepunkteigenschaften bearbeiten“ kann man unten im Dialog Bedingungen an den Haltepunkt knüpfen, ohne den Code zu ändern
- Es gibt mehrere Fenster in dem man Variablen ansehen (und auch verändern!) kann
 - Beobachten: Hier kann man auf der linken Seite Variablen eintragen
 - Auto: Hier sind automatisch Vorschläge für Variablen enthalten
 - Natürlich sind nur Variablen möglich, die an dieser Stelle im Code bekannt sind!

Aufrufliste (Call stack)

- In der Aufrufliste kann man sehen, von wo die Funktion in der der Haltepunkt ist, aufgerufen wurde
- Mit einem Doppelklick kann man in die Funktion springen, die aufgerufen hat – dann kann man sich auch die Variablen ansehen, die in der Funktion bekannt waren

Ändern und Kompilieren (Edit and Compile)

- Das VS bietet die Möglichkeit, das Programm zu unterbrechen, eine Änderung vorzunehmen, zu kompilieren und *an der unterbrochenen* Stelle fortzufahren!
- Das ist aus technischen Gründen nicht immer möglich, aber kann sehr hilfreich sein

Ein Schritt zurück...

- ... wäre der Traum aller Programmierer, aber ist technisch nicht korrekt realisierbar
- Wenn man den gelben Pfeil im Debugger hochzieht, wird *nicht* der Programmfluss (mit Variablen) zurückgespult, sondern das Programm führt als nächsten Befehl den Befehl nach der Pfeilspitze aus