

# Einführung in C++ mit Microsoft VS

# Gliederung

- Allgemeines zu C++ und Unterschiede zu Java
  - Header-Files
  - Zeiger/Strukturen
  - Namespaces
  - 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 nicht statische 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

## Einfache Datentypen

- **int**: ganze Zahl
- **unsigned int**: natürliche Zahl
- **bool**: Wahrheitswert
- **double, float**: Gleitkommazahl
- **char**: Byte, Zeichen

```
int rufnummer;  
rufnummer = 28917058;  
  
double temperatur = 10.3;  
  
bool sommerzeit = true;
```

# Strukturen und Aufzählungen

- **Array** : Zusammenfassung von Elementen desselben Typs

```
int rufnummern[10];
rufnummern[0] = 28917058;
```

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

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

```
TelefonbuchEintrag.Nachname = „Ryu“;
TelefonbuchEintrag.Vorname = "Hilla";
TelefonbuchEintrag.Vorwahl = 089;
TelefonbuchEintrag.Rufnummer = 28917058;
```

- **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. Memberdatendeklaration

```
int m_number;
```

7. Präprozessoranweisungen

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

# Die Header-Datei

myClass.h

```
class MyClass : MyParentClass
{ // die Klasse MyClass erbt von MyParentClass
public: // jeder kann auf diese Funktionen / Daten zugreifen
    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: // Zugriff innerhalb der Klasse und durch abgeleitete Klassen
    virtual int fun(); //eine virtuelle Funktion

private: // Zugriff nur innerhalb der Klasse
    void fu(); //eine Funktion
    std::string m_someString; //eine Membervariable
}
```

# Die Implementierung

myClass.cpp

```
#include "myClass.h"

MyClass::MyClass(){
    m_someNumber = 5;
    m_someString = "Eine Intialisierung vom Text";
} //Standardkonstruktor

MyClass::MyClass(std::string text){
    m_someNumber = 5;
    m_someString = text;
} //zweiter Konstruktor

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

void MyClass::fu(){} //eine Funktion
int MyClass::fun(){return m_someNumber;} //eine virtuelle Funktion
double MyClass::funct(){return 2.0;} //eine statische Funktion
```

## Warum Zeiger?

```
Telefonbuch t;  
string suchbegriff;  
t.neuerEintrag(...);  
t.neuerEintrag(...);  
...  
Eintrag e;  
e = suche(suchbegriff, t);
```

- Funktionsparameter und Rückgabewerte können Geschwindigkeit beeinträchtigen

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

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

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

# Zeiger und Referenz

## Ein Zeiger (MyClass\*)

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

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

## Eine Referenz (MyClass&)

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

JAVA kennt nur die Mischung aus beiden

# Umwandlung

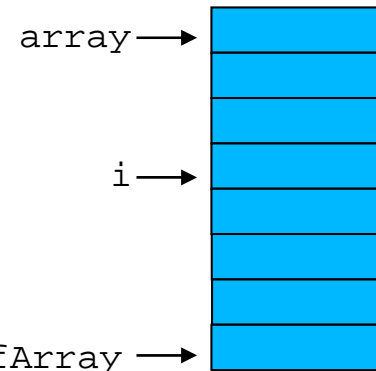
- Die gleichen Zeichen werden bei Umwandlungen benutzt
- `&` (Adress Operator) vor einem Objekt bedeutet „Ich hätte gern den Speicherort vom Objekt“
- `*` (Dereference Operator) vor einem Zeiger (bzw. Speicherort) bedeutet „Ich hätte gern die Referenz (bzw. das Objekt) zum Zeiger“

```
MyClass object;  
MyClass* objectPointer;  
  
objectPointer = &object;  
object = *objectPointer;
```

## 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:

```
MyClass mol; // Stack  
MyClass* mo2 = new MyClass(); // 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 kommen alle Objekte auf den Heap; der GarbageCollector gibt es dann wieder frei



Fehlerquelle  
Nr. 1 !

## Tipps zu Zeigern (Pointer) und Referenzen

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

```
MyClass* object = new MyClass();  
:  
delete object;
```

- Man sollte sich angewöhnen, alle Pointer mit 0 zu initialisieren und vor `delete` auf 0 zu überprüfen

```
MyClass* object = 0;  
if(object)  
    delete object;
```

- Zu `new MyClass[42]` gehört `delete[ ]`

## const Deklaration

- Funktion, bei der die Parameterwerte nicht verändert werden

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

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

```
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 MyClass*` ist ein Pointer auf ein konstantes Objekt. Der Pointer kann geändert werden, muss aber auf ein konstantes Objekt zeigen.
  - `MyClass const*` ist ein konstanter Pointer auf ein Objekt. Der Pointer ist konstant kann aber auf konstante und nicht-konstante Objekte zeigen.
  - `const MyClass const*` ist ein konstanter Pointer auf ein konstantes Objekt.

## Coding Guidelines - Funktionen

- Eine Funktion sollte nicht mehr als eine Seite beinhalten
- Die Funktionalität des Codes sollte durch Kommentare beschrieben sein
- Alle öffentlichen und geschützten Funktionen und Variablen sollten Kommentare im Header haben
- Variablendeklaration sollte in der Nähe des ersten Aufrufs sein
- Destruktoren sollten immer virtuell sein (`virtual ~MyObject` im Header)

## Coding Guidelines - Namen

- Der Sinn sollte erkennbar sein (computeArea, width, result)
- Keine zu ähnlichen Namen (myReference, myRefer)
- 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

## Coding Guidelines - Datenfluss

- Const benutzen
- Es gibt keinen vernünftigen Grund für globale Variablen, Funktionen oder anderes
- Man muss nicht immer alles mit Vererbungshierarchien lösen – nur wenn es die Aussage trifft

## Coding Guidelines - Fehlersuche

- Din A4 Blatt mit seinen Lieblingsfehlern anlegen
- Compiler-Warnungen haben ihren Sinn!
- Linkerfehler haben 3 Hauptursachen
  - 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 neu erfinden
- STL (Standard Template Library) sollte man kennen und auch verwenden
  - `std::vector` (`std::list`, `std::map`) statt Arrays
  - Suchalgorithmen gibt es schon
  - `std::string` erlaubt meist bessere Handhabung als `char *`
- Vieles gibt es schon in vernünftiger Qualität mit einer freundlichen Lizenz

## Die Standardbibliothek STL

- Viele elementare Algorithmen und Datenstrukturen sind darin vorimplementiert
- die wichtigsten Header sind vector, set, algorithm, ....

```
include <iostream>
include <vector>
int main() {
    std::vector<int> vec;
    for (int i = 0; i < 5; ++i) {
        vec.push_back(i);
    }
    vec.insert(vec.begin(), 5);
    return 0;
}
```

- Vector enthält am Schluss: 5 0 1 2 3 4

# Standardbibliotheken

- Weitere wichtige Header sind `cmath`, `string`, `iostream`, ...

```
include <iostream>
include <string>

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

# Microsoft VS

# Integrierter Debugger

- Kompilieren im Debugmodus:
  - Der Compiler fügt dem Code zusätzliche Informationen 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)

- VS bietet die Möglichkeit, das Programm zu unterbrechen, eine Änderung vorzunehmen, zu kompilieren und *an der unterbrochenen* Stelle fortzufahren!
- 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 Programmlauf (mit Variablen) zurückgespult, sondern das Programm führt als nächsten Befehl den Befehl nach der Pfeilspitze aus