

Grundlagen der Programmierung

VL 16: Programmieren in C++

Prof. Dr. Samuel Kounev
Jóakim von Kistowski
Norbert Schmitt

Abgrenzung

- Diese Vorlesung ist eine Übersicht über C++ und die Unterschiede zu C und Java
- Wir behandeln den ISO C++ 98 Standard. Moderne Compiler bieten oft Zusatzfunktionen, die einige der Einschränkungen von ISO C++ 98 umgehen
- Weitere Materialien zum C Lernen:
 - Der C++ Primer (siehe Literaturliste in der 1. Vorlesung)
 - Allgemeines Tutorial
 - <http://www.cprogramming.com/tutorial/c++-tutorial.html>

Inhalt

- Kurzüberblick C++: was ist neu im Vergleich zu C?
- Neue syntaktische Details in C++
 - Namespaces
 - C++ Standard Library
- Referenzen
- Klassen
 - Speicherverwaltung von Klassen
 - Vererbung



C++



Java/C#



Ruby



C

- Grundsätzlich: C++ ist eine Erweiterung der Sprache C
 - Jedes gültige C Programm ist auch ein gültiges C++ Programm und kann mit einem C++ Compiler kompiliert werden
- Kleine Geschichte
 - C++ Erfinder: Bjarne Stroustrup
 - 1979: C with Classes
 - Neu: Klassen, einfache Vererbung, default Funktionsparameter
 - „C with Classes“ Compiler übersetzt Code in C
 - 1983: Neuer Name: C++
 - Neu: virtual, Überladen von Funktionen, Referenzen
 - 1985: C++ 1.0 wird als kommerzielle Sprache veröffentlicht
 - 1989: C++ 2.0
 - 1998: Erster C++ Standard (ISO C++ 98), Standard Template Library
 - 2011: ISO C++11
 - 2014: ISO C++14



1. Objektorientierte Konzepte

- Klassen, Vererbung (auch Mehrfachvererbung), Kapselung, Polymorphismus, Function-Overriding und -Overloading, Operator-Overloading

2. Referenzen

- Aliase für ein existierendes Objekt; werden intern typischerweise durch Pointer realisiert
- Werden bei der Parameterübergabe „by reference“ und bei Operator-Overloading benutzt

3. Templates

- Zur „Parametrisierung“ von Klassen oder Funktionen

4. Exceptions

- Zur standardisierten Fehlerbehandlung, ähnlich wie in Java

5. Namespaces

- Ähnliches Konzept wie Packages in Java; vor allem um Namenskollisionen zu vermeiden

6. Syntaktische Details

- Datentyp **bool**, Datentypkonversion mit Funktionsschreibweise `type(...)`, ...

7. C++ Standard Library

- C Standard Library + neue Funktionalität
- **iostream** Library: Ersatz der Input/Output Funktionen aus C (**stdio.h**)
- Standard Template Library (STL): Klassen für Container: vectors, queues, lists, etc.

Hello World – Java, C und C++

```
/* HelloWorld.java */
public class HelloWorld {
    // Hauptprogramm
    public static void main(String[] args) {
        System.out.println("Hello World in Java");
    }
}
```

```
/* helloworld.c */
#include <stdio.h>

/* Hauptprogramm */
int main(void) {
    printf("Hello World in C\n");
}
```

```
/* helloworld.cpp */
#include <iostream>
using namespace std;

// Hauptprogramm
int main() {
    cout << "Hello World in C++" << endl;
}
```

End-of-line character „\n“

cout (= console out) ist ein Objekt aus der **iostream**-Library das die Standard-Ausgabe repräsentiert. Mit dem Ausgabeoperator << ist es möglich, beliebig viele Teile der Ausgabe aneinanderzuhängen, die sich auch auf mehrere Zeilen verteilen dürfen.

Inhalt

- Kurzüberblick C++: was ist neu im Vergleich zu C?

- Neue syntaktische Details in C++

- Namespaces
- C++ Standard Library

- Referenzen

- Klassen

- Speicherverwaltung von Klassen
- Vererbung



C++



Java/C#



Ruby



C

- Zweite Schreibweise für explizite Datentypkonvertierungen
 - `var = (type) expression` // Operator, C/C++
 - `var = type(expression)` // Funktion, nur in C++
- Variablen können überall in einem Block deklariert werden
 - ANSI C: Variablendeklaration nur am Anfang eines Blocks möglich
 - C++: Ermöglicht z.B. die `for(int...)`-Variante:

```
// ANSI-C
int i;
for (i = 0; i < 10; i++) {
    sum += i;
}
```

```
// C++
for (int i = 0; i < 10; i++) {
    sum += i;
}
```

- **Default-Werte** für Argumente von Funktionen
- Beim Aufruf müssen nicht alle Argumente explizit angegeben werden
- Nicht angegebene Argumente erhalten Default-Wert
- Argumente mit Default-Werten müssen nach allen anderen Argumenten folgen

■ Beispiel:

```
...  
// A function calculating the volume of a cube  
  
// Function declaration:  
int volume (int a, int b = 2, int c = 4);  
// Default values have to be stated on declaration, not definition!  
  
// Function definition:  
int volume (int a, int b, int c) {  
    return a * b * c;  
}  
  
// Function usage:  
int res1 = volume(3, 1, 5); // res1 = 15  
int res2 = volume(3, 3);    // c = 4, res2 = 36  
int res3 = volume(3);       // b = 2, c = 4, res3 = 24  
int res4 = volume();        // Compiler error!  
...
```

- Namespaces erlauben Strukturierung von Programmcode
 - Erlaubt u.a. Unterscheidung von Klassen mit gleichen Namen
- Default-Namespace: Globaler Namespace
 - Kleinere Programme verwenden meist den globalen Namespace
- Code aus einem anderen Namespace mit Scope-Resolution-Operator ::
 - Beispiel: **mynamespace::MyStruct** → Struktur **MyStruct** aus dem Namespace **mynamespace**

```
using namespace mynamespace;
```

Imports all names from **mynamespace** into the current (usually global) namespace.

- Header der C++ Standard Library haben keine Endung .h
 - `#include <iostream>`
- Alte C-Header erhalten ein vorangestelltes „c“
 - `stdio.h` → `cstdio`
 - `string.h` → `cstring`
 - ...
- Die Standard Library ist im Namespace `std` definiert
 - Zugriff mit `std::`
 - `std::cout << "Hello World in C++" << std::endl;`
- Alternativ mit `using namespace std;`
 - Importiert alle Namen des Namespace `std` in den aktuellen (meist den globalen) Namensraum
 - `cout << "Hello World in C++" << endl;`

- Kurzüberblick C++: was ist neu im Vergleich zu C?
- Neue syntaktische Details in C++
 - Namespaces
 - C++ Standard Library

Referenzen

- Klassen
 - Speicherverwaltung von Klassen
 - Vererbung



C++



Java/C#



Ruby



C

- Eine **Referenz** ist ein Alias für ein existierendes Objekt
 - Object ist hier allgemein gemeint, d.h. nicht nur Instanzen von Klassen sondern auch primitive Datentypen und Strukturen
- Deklaration und Initialisierung:

```
int i = 5;
int &ir1 = i;    // ir1 is a reference (an alias) for i
int &ir2 = ir1;  // ir2 is an alias for ir1, and
                // therefore also an alias for i
int &ir3 = i;    // ir3 is an alias for i
int &ir4;        // Compiler error, ir4 must be initialized!
```

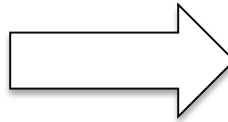
- Wird eine Referenz in einem Ausdruck verwendet, dann wird implizit nicht die Referenz selbst, sondern immer das referenzierte Objekt verwendet

```
int i = 5;
int &j = i;      // j is an alias for i
j++;            // executes i++; i (and j) are therefore 6
i += 3;         // i = 9; j = 9 as well
int k = j;      // executes k = i, therefore k = 9, j is not an
                // alias of k
```

- Referenzen sind intern als Pointer implementiert, es gibt aber drei wichtige Unterschiede zwischen Referenzen und Pointern
 1. Eine Referenz muss **immer** ein gültiges Objekt referenzieren
 2. Eine Referenz kann nicht so verändert werden, dass sie ein anderes Objekt referenziert
 3. Wenn von einer Referenz auf das Objekt zugegriffen wird, wird kein expliziter Dereferenzierungsmechanismus benötigt

- Vom Compiler werden Referenzen mit Pointern implementiert:
 - wenn eine Referenz **p** verwendet wird, wird **p** durch ***p** ersetzt
 - wenn eine Referenz mit **a** initialisiert wird, wird **a** durch **&a** ersetzt
- Beispiel:

```
int i = 5;  
int &j = i;  
j++;  
i += 3;  
int k = j;
```



```
int i = 5;  
int *j = &i;  
(*j)++;  
i += 3;  
int k = (*j);
```

- Durch die Verwendung von Referenzen wird der Code übersichtlicher und weniger fehleranfällig

- Wo werden Referenzen gebraucht?
 - Parameterübergabe → Call-by-Reference
 - Operator Overloading (wird hier nicht behandelt)
- Call-by-Reference, Beispiel:

```
int main() {  
    int a = 5;  
    increment(a); // a = 6  
}  
  
void increment(int &a) {  
    a++;  
}
```

- Referenzen bieten eingeschränkte Funktionalität
 - (Leider) kein kompletter Ersatz für den Gebrauch von Pointern...

Inhalt

- Kurzüberblick C++: was ist neu im Vergleich zu C?
- Neue syntaktische Details in C++
 - Namespaces
 - C++ Standard Library

■ Referenzen

- **Klassen**
 - Speicherverwaltung von Klassen
 - Vererbung



C++



Java/C#



Ruby



C

- C++ bietet Klassen an: Effektiv **structs**, welche zusätzlich Methoden enthalten können
- Anders als in Java wird die Verwendung von Klassen nicht erzwungen
 - d.h. Methoden können auch außerhalb von Klassen auftreten
- Bei Klassen, Attributen und Methoden gilt auch das Prinzip der **Vorwärtsdeklaration!**
 - Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig
 - Lösung: Prinzipielle separate Deklaration in Header-Dateien

- Prinzip der **Vorwärtsdeklaration**
 - Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig.
 - Beispiel: Inkrementiert und verdoppelt Wert n -Mal. Enthält zyklische Abhängigkeit.

```
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
int increment(int a, int i){  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
int main() {  
    int i = 5;  
    return increment(1, i);  
}
```

- Compilerausgabe:

```
main.cpp: In function 'int doubleValue(int, int)':  
main.cpp:2:27: error: 'increment' was not declared in this scope  
    return increment(2*a,i);  
                   ^
```

- Lösung: Separate Vorwärtsdeklaration

Vorwärtsdeklaration

Deklariert `increment(int,int)` vor der Implementierung.

Deklarationen werden normalerweise in separaten **header-Dateien** (.h) abgelegt die mit **#include** eingebunden werden (z.B. **#include** <string.h> bindet die `string.h` header-Datei aus der C-Standard-Library ein)

```
int increment(int, int);

int doubleValue(int a, int i) {
    return increment(2*a, i);
}

int increment(int a, int i){
    if (i == 0) { return a; }
    return doubleValue(a+1, i-1);
}

int main() {
    int n = 5;
    return increment(1, i);
}
```

Referenzierung von deklarierter Funktion

`doubleValue(int,int)` kann `increment(int,int)` referenzieren, da es bereits deklariert worden ist.

Auflösung der separaten Implementierung

Als Teil des Kompiliervorgangs ordnet der Linker die Implementierung von `increment(int,int)` der Deklaration zu.

ACHTUNG: Das geht auch Datei-übergreifend!

- Enthalten Deklarationen von Funktionen, Datenstrukturen und Klassen
 - Bei Klassen: Attributen und Methoden
- **private** / **public** Modifizierer können für ganze Deklarations-Blöcke verwendet werden

File myclass.h

```
// class declaration
class MyClass {
    private:
        int x;
    public:
        int getX();
        void setX(int x);
};
```

- Include-Guards verhindern das mehrfache Einbinden eines Headers

Preprocessor directives:

<if-not-defined> *identifier* ←

<define> *identifier* ←

Preprocessor directives are lines preceded by a hash sign (#). These lines are not program statements, but directives for the preprocessor. The **preprocessor** examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

(<http://www.cplusplus.com/doc/tutorial/preprocessor>)

File myclass.h

```
//include guard
#ifndef MYCLASS_H
#define MYCLASS_H

//Class declaration
class MyClass {
    private:
        int x;
    public:
        int getX();
        void setX(int x);
};

#endif
```

Code will be compiled on the first inclusion of the header file

- Enthalten die Implementierungen der Methoden
- Können auf die Variablen aus dem Header zugreifen
- :: Notation zeigt Zugehörigkeit zur Klasse an

File container.cpp

```
#include "myclass.h"
```

```
int MyClass::getX() {  
    return x;  
}
```

```
void MyClass::setX(int x) {  
    (*this).x = x;  
}
```

Die **getX** Methode von
der Klasse **MyClass**

Achtung:


Anders als in Java ist **this**
ein Pointer auf das Objekt,
nicht das Objekt selbst!

- Objekte können direkt deklariert und angesprochen werden
 - Anders als in Java, werden hier die entsprechenden Variablen ähnlich wie Variablen einfacher Datentypen behandelt
 - Die Variablen sind KEINE Referenzen oder Pointer; sie beziehen sich auf das Objekt selbst!
 - Members werden per Punktnotation angesprochen

```
#include <iostream>
#include "myclass.h"
```

```
int main() {
    MyClass myclass = MyClass();
    myclass.setX(5);
    return 0;
}
```

Standardkonstruktor wird für die Instanziierung verwendet



- Der **->** Operator
 - Einfacher Zugriff auf Members von Objekt- oder struct-Pointern
 - Für jedes Objekt/struct **x** mit Pointer **ptr = &x** gilt:

`ptr->y;` ist äquivalent zu `(*ptr).y;`

```
#include <iostream>
#include "myclass.h"

int main() {
    MyClass myclass = MyClass();
    myclass.setX(5);
    MyClass *ptr = &myclass;
    cout << ptr->getX();
    // equivalent to: cout << (*ptr).getX();
    return 0;
}
```

- Jedes Programm wird in drei Speicherbereiche (Segmente) aufgeteilt:
 - *text segment* (oder *code segment*): Programminstruktionen und Konstanten
 - *stack segment*: Lokale Variablen und Parameter bei Funktionen
 - *heap segment*: Globale Variablen und dynamisch allozierter Speicher (im Code)
- Direkte Speichermanipulation mit Pointern ist möglich → **Mögliche Speicherfehler!**
- Explizite **dynamische Allokation** von Speicher mit
 - `void * malloc(int size)`: Allokiert `size` zusammenhängende Bytes. Gibt einen Pointer auf die Adresse des ersten Bytes zurück.
 - Größe von erforderlichem Speicher kann mit `sizeof(<Datentyp>)` ermittelt werden
 - `void free(void * ptr)`: Deallokiert den Speicherblock, der bei `ptr` beginnt.
 - Aufräumen von ungenutztem Speicher nicht vergessen!
 - **Keine garbage collection (wie in Java)!**

- Direkte Objekt-Deklaration geschieht auf dem Stack. Objekte sind nur für ihren Code-Block instanziiert.
 - Danach zeigen Pointer auf die Objekte ins Nirvana!
- Allokation auf dem Heap mit **malloc()** und **free()**
- **Oder einfacher:** mit **new** und **delete**
 - **new**: Allokiert Speicher, führt den Konstruktor aus und liefert einen Pointer zurück
 - Beispiel: **MyClass *ptr = new MyClass();**
 - **delete**: Deallokiert Speicher des Objekts, führt den Destruktor aus
 - Beispiel: **delete(ptr);**

- Destruktoren: Werden beim Löschen aufgerufen
 - Aufgabe: Aufräumen des auf dem Heap allokierten Speichers
 - Syntax: `~<Klassenname>();`
- Beispiel

```
//wrapper.h
#include "myclass.h"

class Wrapper {
    private:
        MyClass *myclass;
    public:
        Wrapper();
        ~Wrapper();
        int getX();
        void setX(int x);
};
```

```
//wrapper.cpp
#include "wrapper.h"

Wrapper::Wrapper() {
    myclass = new MyClass();
}

int Wrapper::getX() {
    return myclass->getX();
}

void Wrapper::setX(int x) {
    myclass->setX(x);
}

Wrapper::~~Wrapper() {
    delete(myclass);
}
```

- Klassen können mit Hilfe des `:` Operators von anderen Klassen erben
- Members der Elternklasse sind dann je nach Sichtbarkeit direkt zugänglich (private, protected, public)

Inheritance operator
Like “extends” in Java

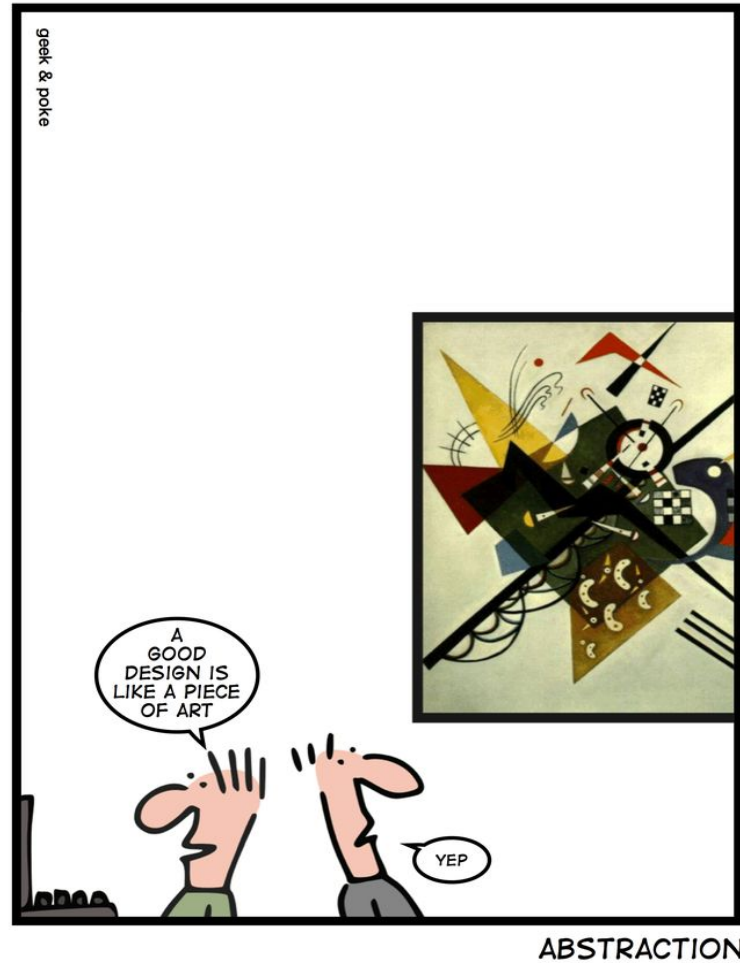
```
class A {  
    public:  
        int getDataA() {  
            return 1;  
        }  
};
```

```
class B : A {  
    public:  
        int getDataB() {  
            return 2;  
        }  
};
```

- Interfaces gibt es nicht. Sie werden i.A. direkt durch Header-Dateien definiert
 - Header sind nicht das gleiche wie Interfaces! Header deklarieren u.a. private Members, Interfaces in Java nicht
 - Eine Möglichkeit zu erzwingen, dass mehrere Klassen die gleichen Methoden implementieren, ist die Klassen von einer gemeinsamen abstrakten Mutterklasse erben zu lassen
 - Dieser Ansatz ist allerdings weniger flexibel als **interfaces** in Java

- Vorlesung „Fortgeschrittenes Programmieren“ - „Advanced Programming“ (Master)
 - C++ References
 - Polymorphie (Polymorphismus)
 - Schlüsselwort virtual
 - Function-Overriding und -Overloading, Operator-Overloading
 - Mehrfachvererbung
 - Diamond-Inheritance
 - Templates
 - Container-Klassen
 - ...

Fragen?



Danksagung

- Vorlesungsmaterialien von Prof. Dr. Oliver Hummel wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:
Jóakim v. Kistowski