HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 14

Programmieren in C++

Zuletzt haben wir gelernt…

- Was Sockets sind.
- Was Client-Server-Kommunikation bedeutet.
- Wie man die Klasse TcpClient einsetzt.
- Wie man einen Zeitclient und –Server aufbaut.
- Wie man E-Mails über SMTP versenden kann.
- Wie Web-Ressourcen mit dem HttpClient abgerufen werden.
- Die Implementierung des Speisencheckers.
- Was Restful-Services sind.
- Wie man JSON-Daten in C# verarbeiten kann.
- Wie man Fussball-Daten von der OpenLigaDB abruft.

Inhalt heute

- Vorteile von C# und C++
- Klassen in C++
- Speicherverwaltung
- Objekterzeugung
- Konstruktor und Destruktor
- Zuweisungen und der Copy-Konstruktor
- Parameterübergabe
- Präpropzessor, Compiler, Linker
- Makefiles

Vorteile von C#

- In den letzten Semestern haben wir in der Programmiersprache C# programmiert.
  - Für moderne Anwendungen (Desktop, Apps, Web, ..) ist dies die bessere Wahl, da man sehr produktiv arbeiten kann.

- C# hat gegenüber C++ einige Vorteile:
  - Durchgängig objektorientiert, z.B. Schnittstellen und keine Mehrfachvererbung
  - Mit der .Net-Plattform steht ein riesiges Ökosystem zur Verfügung
  - Keine Zeiger, keine manuelle Speicherverwaltung
  - Keine historisch gewachsene Syntax (z.B. Makros, ...)
  - Kein so komplexes Build-System (Präprozessor-Direktiven)

Vorteile von C++

- Wenn eingebettete Systeme entwickelt werden sollen, kann C# nicht genutzt werden.
  - Der MSIL-Code wird durch die .Net Runtime Umgebung ausgeführt.
  - Dies kostet Perfomance, die man in Realtime-Anwendungen nicht hat.

- Daher muss in einbetteten Systemen eine Programmiersprache gewählt werden, die direkt in Maschinencode übersetzt wird.
  - Meist wird dann C/C++ genutzt.

- Mit C/C++ kann sehr hardwarenah programmiert werden.
  - *C macht es einfach, sich selbst ins Bein zu Schießen; C++ erschwert es, aber wenn es dir gelingt, bläst es dir das ganze Bein weg (*Bjarne Stroustrup) → Vorsicht ist geboten!

Klassen in C++

- Auch in C++ existiert das Konzept der Objekte als Exemplare von Klassen.
  - Objekte haben einen eigenen Zustand, den Wert ihrer Variablen.
  - Entsprechend sind viele Dinge zumindest ähnlich: Geheimnisprinzip, Vererbung, Polymorphie, Konstruktoren, ...
  - Es gibt in C++ allerdings keine Eigenschaftsmethoden, wie in C#.
- C++ zwingt den Programmierer nicht dazu, objektorientiert zu programmieren.
  - Funktionen können, wie in C, außerhalb von Klassen definiert werden.
  - Viele Funktionen der C++ Standardbibliothek sind entsprechend nicht Teil einer Klasse.
- Klassen werden in C++ in zwei Teile aufgeteilt:
  - Die Definition (als sog. Header-Datei mit der Dateiendung h bzw. hpp).
  - Die Implementierung der Methoden (Datei mit der Dateiendung cpp).

Header des Bankkontos in C++

```
#ifndef Bankkonto_hpp
#define Bankkonto_hpp

#include <stdio.h>

class Bankkonto {
private:
    double kontostand;

public:
    Bankkonto();
    void Einzahlen(double betrag);
    void Auszahlen(double betrag);
    double GetKontostand();
};

#endif
```

Präprozessoranweisungen sorgen dafür, dass die Klasse im Projekt nur einmal definiert wird.

Variablen und Methoden werden in den Bereichen private, public oder protected definiert.

Im Header werden nur die Prototypen der Methoden definiert.

Implementierung der Klasse

```
#include "Bankkonto.hpp"

Bankkonto::Bankkonto() {
    kontostand = 0;
}

void Bankkonto::Einzahlen(double betrag) {
    kontostand += betrag;
}

void Bankkonto::Auszahlen(double betrag) {
    if (kontostand > betrag)
        kontostand -= betrag;
    else
        throw "Betrag nicht ausreichend";
}

double Bankkonto::GetKontostand() {
    return kontostand;
}
```

Einbinden des Headers, sonst kennt der Compiler die Klasse Bankkonto gar nicht.

Jede einzelne Methode wird nun implementiert.

Speicherverwaltung

- In C++ hat man als Programmierer großen Einfluss auf die Speicherverwaltung.
  - z.B. kann ein Objekt auf dem **Stack** oder dem **Heap** erzeugt werden.

- Objekte auf dem **Stack** werden beim Verlassen des Gültigkeitsbereichs (meist die aktuelle Funktion) automatisch gelöscht.
  - Allerdings ist der Stackspeicher recht klein.

- Objekte auf dem **Heap** müssen manuell wieder gelöscht werden.
  - Der Heap-Speicher entspricht der Größe des Hauptspeichers.

Objekterzeugung

- Auf dem Stack:

```
Bankkonto a;
a.Einzahlen(100);
cout << "Kontostand: " << a.GetKontostand() << endl;
```

- Auf dem Heap:

```
Bankkonto *b = new Bankkonto();
b->Einzahlen(100);
cout << "Kontostand: " << b->GetKontostand() << endl;
delete b;
```

Achtung: b ist ein Zeiger!
Der Speicher muss mit delete wieder freigegeben werden!

Konstruktor

- Auch in C++ werden Objekte mit dem Konstruktor initialisiert.

```
Bruch::Bruch(int z, int n) {
     zaehler = z;
     nenner = n;
}
```

- Existiert ein Konstruktor mit Parametern, kann auch in C++ kein Objekt mehr erzeugt werden, ohne dass entsprechende Werte übergeben werden.

```
Bruch a(1,3); // auf dem Stack
Bruch *b = new Bruch(1,3); // auf dem Heap
```

Destruktor

- Das Gegenstück zum Konstruktor ist der **Destruktor**:

```
Bruch::~Bruch() {
}
```

- Der Destruktor kommt am Ende der Lebenszeit eines Objektes zum Einsatz.
  - Er wird nicht direkt aufgerufen.

- Bei Stack-Objekten wird der Destruktor dann aufgerufen, wenn das Objekt automatisch zerstört wird.
  - Bei Heap-Objekten wird der Destruktor durch die `delete`-Anweisung aufgerufen.

- In C++ ist der Destruktor **viel wichtiger** als in C#.
  - Wenn ein Objekt mit `new` Speicher angefordert hat, muss dieser im Destruktor wieder frei gegeben werden.

Zuweisungen

- Objekte können in C++ auf dem Stack oder auf dem Heap existieren.

- Bei einer Zuweisung wird von einem Stack-Objekt eine echte Kopie erzeugt.
  - Der gesamte Zustand (auch private Member) werden kopiert.

```
Bankkonto a;
a.Einzahlen(100);
Bankkonto b = a;
```

Das Objekt in b ist eine Kopie von a. Alle Änderungen an b haben aber keine Auswirkungen auf a.

- Bei einem Objekt auf dem Heap wird bei einer Zuweisung nur der Zeiger kopiert.
  - Der zweite Zeiger zeigt immer noch auf das selbe Objekt.

```
Bankkonto *a = new Bankkonto();
Bankkonto *b = a;
```

Die Zeiger a und b zeigen beide auf das selbe Objekt.

Copy-Konstruktor

- Damit bei einer Zuweisung eines Stack-Objektes eine Kopie erzeugt werden kann, existiert ein besonderer Konstruktor.
  - Der **Copy-Konstruktor**.
  - Er existiert immer bereits in einer einfachen Variante.
  - Der Wert aller Objektvariablen wird dabei in das Ziel kopiert.

- Für erweiterte Fälle kann dieser aber ersetzt werden.
  - Wenn Objekte z.B. größere Mengen Speicher reserviert haben, macht dies Sinn.

- Es muss dann der folgende Konstruktor eingeführt werden:
  - `Bankkonto::Bankkonto(const Bankkonto &k);`

Parameterübergabe

- Parameter (z.B. Objekte) können in C++ auf drei verschiedene Arten an Funktionen übergeben werden.
  - Die Übergabemethode wird durch die Art des Parameters in der Definition der Funktion bestimmt.

- **Übergabe als Wert**
  - Bei der Übergabe wird mit Hilfe des sog. **Copy-Constructors** eine Kopie des Objektes erzeugt.
  - `void alsWert(Bankkonto z) { }`

- **Übergabe als Zeiger**
  - Alle Änderungen schlagen auch auf das ursprüngliche Objekt durch.
  - `void alsZeiger(Bankkonto *z) { }`

- **Übergabe als Referenz**
  - Alle Änderungen schlagen auch auf das ursprüngliche Objekt durch.
  - In C# ist dies die einzig mögliche Variante.
  - `void alsReferenz(Bankkonto &z) { }`

https://repl.it/@loswochos/ParametersCpp

HOCHSCHULE HAMM-LIPPSTADT

Übergabearten

```
#include <iostream>
#include "Bankkonto.hpp"

using namespace std;

void alsWert(Bankkonto z) { z.Einzahlen(100); }
void alsZeiger(Bankkonto *z) { z->Einzahlen(100); }
void alsReferenz(Bankkonto &z) { z.Einzahlen(100); }

int main(int argc, const char * argv[]) {
    Bankkonto a;

    alsWert(a);
    alsZeiger(&a);
    alsReferenz(a);

    cout << "Kontostand: " << a.GetKontostand() << endl;

    return 0;
}
```

Welchen Kontostand hat a am Ende?

Operatorenüberladung

- Bruchobjekte können wir zunächst nicht addieren, wie normale Zahlen.
  - Der Operator „+“ ist auf den Objekten nicht definiert.

- In C++ (auch in C#) kann man Operatoren überladen.
  - Dadurch kann man Bruchobjekte addieren, wie normale int-Werte:
  - `Bruch c = a + b; // a und b sind auch Brüche.`

- Für jeden Operator, der auf Objekten angeboten werden soll, muss eine besondere Methoden implementiert werden.
  - `Bruch operator+(const Bruch &b);`

Präprozessor

- Für die Verarbeitung aller Anweisungen, die mit dem Doppelkreuz # beginnen, ist in C++ der sog. **Präprozessor** verantwortlich.
  - Der Präprozessor wird auf den Quellcode angewendet, bevor der Compiler das Programm übersetzt.

- Der Präprozessor wird in C++ benötigt, um Projekte aus mehreren Dateien zusammenzusetzen.
  - Die wichtigste Anweisung ist sicherlich `#include`.
  - Damit wird eine Datei (meist ein Header) in die aktuelle Datei eingebunden.
  - Erst dadurch kennt der Compiler den Namen von Typen (Klassen).

- Es existieren noch weitere Anweisungen für den Präprozessor, die hilfreich sein können.

Präprozessoranweisungen

Anweisung	Bedeutung	Beispiel
#include	Bindet eine Datei ein.	#include "funktionen.h"
#define	Definiert ein Symbol.	#define DEBUG

#ifdef
#endif
	Führt einen Abschnitt im Programmcode ein, der nur dann übersetzt wird, wenn ein Symbol definiert ist.	#ifdef DEBUG
		printf("Im Debug-Modus!\n");
		#endif

#ifndef
#endif
	Führt einen Abschnitt im Programmcode ein, der nur dann übersetzt wird, wenn ein Symbol nicht definiert ist.	#ifndef DEBUG
		printf("Nicht im Debug-Modus!\n");
		#endif

Vor doppelter Inklusion schützen

- Wenn ein Programm aus vielen Dateien besteht, kann es sein, dass eine bestimmte Header-Datei an mehreren Stellen eingebunden wird.
  - Dies führt dazu, dass eine Funktion doppelt deklariert wäre.
  - Dies ist nicht erlaubt.
  - Der Compiler bricht dann den Übersetzungsvorgang ab.

- Mit Hilfe der Anweisungen des Präprozessors können wir diese doppelte Inklusion von Funktionsprototypen verhindern.
  - Die Anweisungen für den Präprozessor sorgen für bedingte Übersetzung, d.h. ein gewisser Teil wird nur unter bestimmten Bedingungen übersetzt.

Header-Datei mit Schutz vor mehrfacher Inklusion

- Die Präprozessoranweisungen `#define` und `#ifndef` helfen uns nun, eine Header-Datei zu präparieren.
  - Eine Header-Datei mit Schutz vor mehrfacher Inklusion sieht wie folgt aus:

```
#ifndef FUNKTIONEN_H_
#define FUNKTIONEN_H_

void arrayAusgabe(int array[], int anzahl);

#endif
```

- Achtung: Die meisten Entwicklungsumgebungen erzeugen diese Anweisungen bereits automatisch, sobald eine neue Klasse zum Projekt hinzugefügt wird.

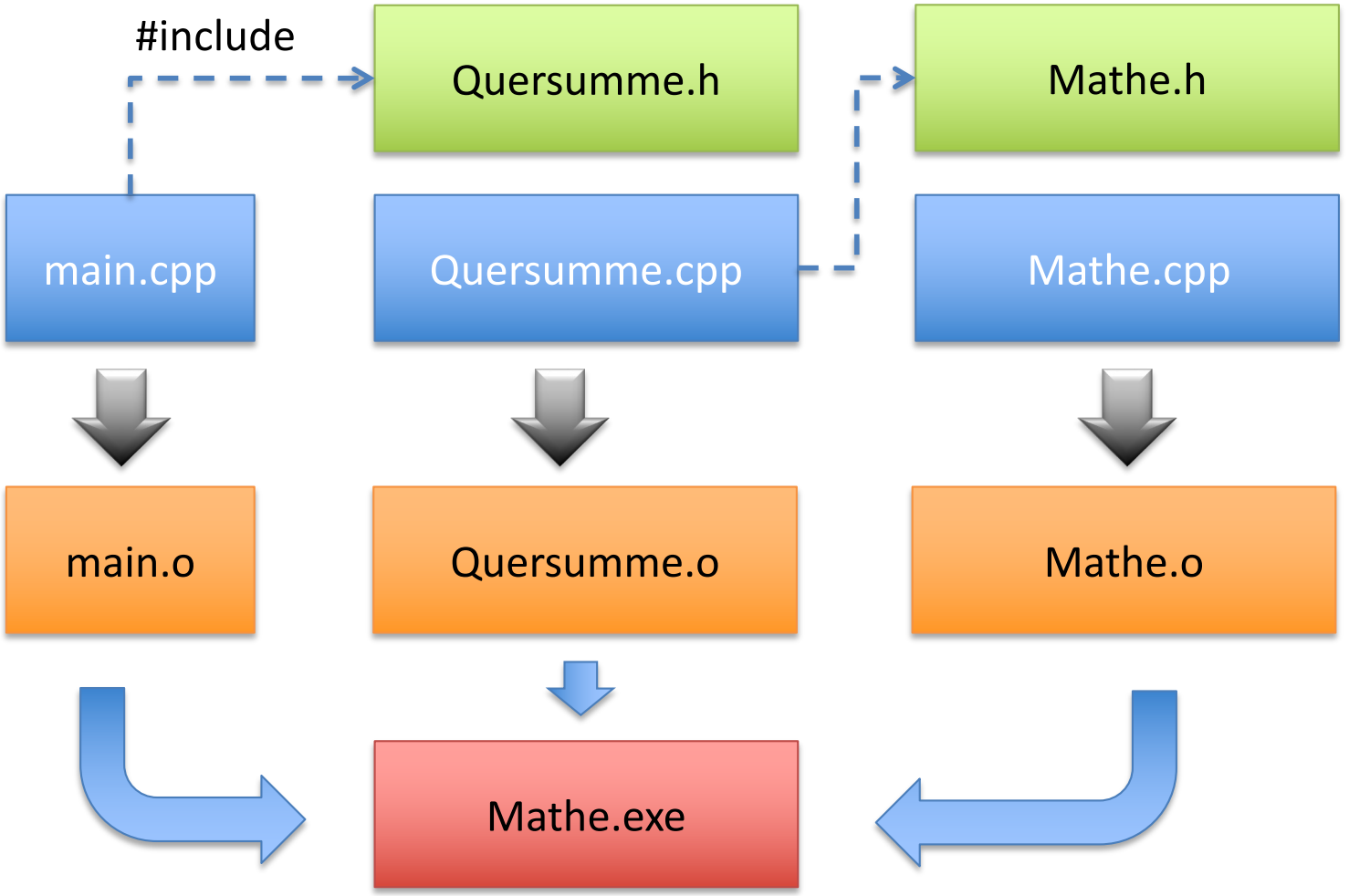
Präprozessor, Compiler, Linker

- **Präprozessor**
  - Wird als erstes noch vor dem Compiler aufgerufen.
  - Verarbeitet alle Anweisungen mit einem # am Anfang.
  - Bindet z.B. die Prototypen der Funktionen aus den Header-Dateien ein.

- **Compiler**
  - Übersetzt den Quellcode jeder Quellcodedatei "cpp" in eine Objektdatei mit der Dateiendung "o".

- **Linker**
  - Bindet alle Objektdateien zusammen (*engl. to link*).
  - Erzeugt dadurch z.B. das ausführbare Programm mit der Endung "exe" (unter Windows).

Projekte aus mehreren Teilen



1. Schritt:
Präprozessor

2. Schritt:
Compiler

3. Schritt:
Linker

Erstellungsprozess

- Wenn Projekte aus vielen Dateien bestehen, sind viele Schritte notwendig, um am Ende zum ausführbaren Programm zu kommen.
  - Übersetzen jeder einzelnen Quelldatei.
  - Binden der Dateien.

- Hierfür gibt es entsprechende Befehle der sog. **GNU Toolchain**.
  - Auf der Konsole ausführbar.

- Übersetzen der Datei Main.cpp in eine Objektdatei:

```
g++ -c Main.cpp –o Main.o
```

- Übersetzen der Datei Funktionen in eine Objektdatei:

```
g++ -c Funktionen.cpp –o Funktionen.o
```

- Binden der beiden Objektdateien zum ausführbaren Programm Main.exe:

```
g++ Main.o Function.o –o Main.exe
```

Makefiles

- Je größer das Projekt, desto mehr Dinge gibt es zu tun.
  - Die Befehle möchte man nicht mehr alle manuell aufrufen müssen.
  - Der Erstellungsprozess sollte am besten automatisiert werden.

- Ein spezielles Script dient der Automatisierung.
  - Ein sog. **Makefile**.
  - Ein Makefile ist eine einfache Textdatei, die von dem Kommando **make** eingelesen wird.

- Das Makefile definiert, aus welchen Quelldateien das Projekt besteht und wie diese voneinander abhängen.
  - Das Makefile definiert, wie die Dateien übersetz werden.
  - Welche Bibliotheken müssen evtl. eingebunden werden?
  - Wie wird das Programm gebunden?

Beispiel Makefile

Das Erstellungsziel *all* sollte immer definiert sein.

Die benötigten Anweisungen für die Übersetzung stehen in der nächsten Zeile nach einem Tabulator!

Hier hängt *all* von den Zielen main.o und Funktionen.o ab. Diese müssen zuerst erstellt werden.

```
all: main.o Funktionen.o
    g++ main.o Funktionen.o -o MakeText.exe

main.o:
    g++ -c main.cpp

Functions.o:
    g++ -c Funktionen.cpp

clean:
    rm -rf *.o MakeTest.exe
```

Die Ziele main.o und Funktions.o definieren, wie diese Objektdateien erzeugt werden.

Wir haben heute gelernt…

- welche Vorteile C# und C++ jeweils haben.
- wie Klassen in C++ erstellt werden.
- welche die manuelle Speicherverwaltung in C++ funktioniert.
- Wie man in C++ Objekte erzeugt.
- Wie man Konstruktor und Destruktor benutzt.
- Welche Eigenarten Zuweisungen in C++ haben und was der Copy-Konstruktor macht.
- Wie Objekte in Form von Parametern übergeben werden.
- Wie die Übersetzung eine C++ Projektes mit dem Präprozessor, dem Compiler und dem Linker funktioniert.
- Wie man mit Makefiles den Übersetzungsprozess steuert.