Objektorientierte Programmierung mit C++
WS 2016/2017

Andreas F. Borchert

Universität Ulm

25. Oktober 2016

Syllabus

Inhalte:

- Einführung in OO-Design, UML und »Design by Contract«.
- Einführung in C++
- Polymorphismus in C++
- Templates
- Funktionsobjekte und Lambda-Ausdrücke in C++
- Dynamische Datenstrukturen mit *smart pointers*
- Statischer vs. dynamischer Polymorphismus
- STL-Bibliothek
- iostream-Bibliothek
- Ausnahmenbehandlungen
- Fortgeschrittene Template-Techniken, Metaprogrammierung
- Potentiale und Auswirkungen optimierender Übersetzer bei C++

C++ ist trotz zahlreicher historischer Relikte (C sei Dank) und seiner hohen Komplexität nach wie vor interessant:

- Analog zu C bietet C++ eine hohe Laufzeit-Effizienz.
- Im Vergleich zu anderen OO-Sprachen kann sehr flexibel bestimmt werden, wieviel statisch und wieviel dynamisch festgelegt wird. Entsprechend muss der Aufwand für OO-Techniken nur dort bezahlt werden, wo er wirklich benötigt wird.
- Programmierung des Übersetzers (Metaprogrammierung).
- Zahlreiche vorhandene C-Bibliotheken wie etwa die BLAS-Bibliothek oder die GMP (GNU Multi-Precision-Library) lassen sich als Klassen in C++ verpacken.
- Die STL und darauf aufbauende Klassen-Bibliotheken sind recht attraktiv.

Somit ist C++ insbesondere auch für rechenintensive mathematische Anwendungen recht interessant.

Syllabus: Fokus auf C++11

- Im August 2011 wurde der aktuelle ISO-Standard für C++ veröffentlicht, ISO 14882-2012, der beachtliche Neuerungen einführt im Vergleich zu dem vorherigen Standard von 2003.
- Im August 2014 wurde C++14 verabschiedet, der einige (eher kleine) Ergänzungen hinzufügte.
- Verschiedene dieser neuen Techniken wurden bereits experimentell durch Teile der Boost-Library eingeführt und sind deswegen schon seit etwas längerer Zeit etabliert wie etwa *smart pointers*.
- Auch unabhängig von der konkreten Sprache C++ sind diese interessant, weil sie u.a. zeigen, wie
  - Elemente funktions-orientierter Programmiersprachen in klassischen objekt-orientierten Sprachen effizient eingebettet werden können (Lambda-Ausdrücke) und wie
  - komplexe dynamische Datenstrukturen auch ohne *garbage collection* elegant verwaltet werden können (*smart pointers*).
- Ein wesentliches Ziel der Vorlesung ist es zu zeigen, wie moderne Sprachtechniken effizient oder sogar effizienzsteigernd eingesetzt werden können.

- Im Open-Source-Bereich gibt es zwei Übersetzer, die hier in Frage kommen: GCC ab mindestens 4.9 oder Clang.
- Im Rahmen der Vorlesung werden wir primär mit GCC 5.2 arbeiten. Auf der Thales steht 5.2 zur Verfügung, im Pool in E.44 unter Debian noch GCC 4.9.2.
- Um den GCC 5.2 auf der Thales nutzen zu können, sollten Sie bei uns gcc51 in der Datei *~/.options* aufnehmen.

- Kenntnisse in
  - einer Programmiersprache (egal welche) und in
  - Algorithmen und Datenstrukturen (Bäume, Hash-Verfahren, Rekursion).
- Freude am Entwickeln von Software und der Arbeit im Team
- Fähigkeit zur selbständigen Arbeitsweise einschließlich dem Lesen von Manualseiten und der eigenständigen Fehlersuche (es gibt keine Tutoren!)

Syllabus: Ziele

- Erwerb von praktischer Erfahrung und soliden Kenntnissen im Umgang mit C++
- Erlernen der Herangehensweise, wie ausgehend von den Anforderungen und dem Entwurf die geeigneten programmiersprachlichen Techniken ausgewählt werden
- Erlernen fortgeschrittener Techniken und der Erwerb der Fähigkeit, diese sinnvoll einzusetzen. Dazu gehören insbesondere Techniken, die von Java nicht unterstützt werden wie etwa statischer Polymorphismus, Lambda-Ausdrücke und Metaprogrammierung.
- Erwerb von Grundkenntnissen über die Implementierungen verschiedener Techniken, so dass das zu erwartende Laufzeitverhalten eingeschätzt werden kann

Syllabus: Struktur

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Dienstag von 10-12 Uhr im Raum E04 in der Helmholtzstraße 22.
- Die Übungen finden an jedem Donnerstag von 16-18 Uhr im Raum E20 in der Helmholtzstraße 18 statt.
- Webseite: `https://www.uni-ulm.de/mawi/mawi-numerik/lehre/wintersemester-20162017/vorlesung-objektorientierte-programmierung-mit-c.html`
- Alle Vorlesungsteilnehmer mögen sich bitte bei SLC für die Vorlesung registrieren.

- Da die Übungen auf einen anderen Vorlesungstermin von mir fallen, hat sich dankenswerterweise Dr. Michael Lehn bereit erklärt, diese zu übernehmen.
- Die praktische Abwicklung der Übungen wird in den ersten Übungen am 27. Oktober vorgestellt.
- Wenn Sie parallel zu dieser Vorlesung gerne auch Systemnahe Software I hören möchten, ist das gerne möglich. Alle Übungsblätter und Lösungen gibt es auch auf der Vorlesungswebseite.
- Es gibt keine formale Vorleistung für die Teilnahme an der schriftlichen Prüfung. Dennoch wird die intensive Teilnahme an den Übungen dringend empfohlen, weil nur dann eine erfolgreiche Teilnahme an der schriftlichen Prüfung zu erwarten ist.

- Es gibt zwei schriftliche Prüfungen am Ende des Semesters in der Prüfungsperiode.
- Die Termine können wir gemeinsam festlegen. Damit wir das nächste Woche tun können, sollten Sie alle bitte ihre anderen Termine sammeln, damit Konflikte vermieden werden können.
- Die Prüfungstermine sind offen, d.h. Sie können auch den zweiten Prüfungstermin wahrnehmen, ohne an der ersten Prüfung teilgenommen zu haben.
- Es wird rechtzeitig vor der ersten Prüfung eine Probeklausur geben, damit Sie sich besser darauf vorbereiten können.

- Im anschließenden Sommersemester 2017 werde ich die Vorlesung *Parallele Programmierung mit C++* anbieten.
- Diese Vorlesung geht ausführlich auf die verschiedenen Möglichkeiten und Techniken der Parallelisierung ein. Dies erfolgt sowohl aus der Perspektive der Architekturen als auch aus der Sicht der Programmierung.
- Es geht in der Vorlesung darum, die Grundlagen dafür zu erlenen, die es erlauben, geeignete Architekturen für parallelisierbare Problemstellungen auszuwählen und dazu passende Algorithmen zu entwickeln.
- Hierfür ist ebenfalls C++ im besonderen Maße geeignet. Seit C++11 sind insbesondere auch Threads Bestandteil des Standards.

- Die Vorlesungsfolien und einige zusätzliche Materialien werden auf der Webseite der Vorlesung zur Verfügung gestellt werden.
- Dort finden sich auch Verweise auf zwei Arbeitsfassungen des C++-Standards (ISO/IEC 14882):
  - August 2010: Letzte Arbeitsfassung vor C++11.
  - Oktober 2013: Diese Arbeitsfassung wurde zur Grundlage von C++14.
  - Juli 2016: aktuelle Arbeitsfassung für C++17, wird teilweise von GCC 6.x unterstützt.
- Die Standards können im Original als PDF von ISO oder ANSI bezogen werden, sind jedoch dort leider sündhaft teuer.

- Bjarne Stroustrup, *The C++ Programming Language*, ISBN 0-321-56384-0 (vierte Auflage, die C++11 berücksichtigt)
- Bjarne Stroustrup, *Programming: Principles and Practice Using C++*, ISBN 978-0-321-99278-9 (zweite Auflage, berücksichtigt C++14)
- David Vandevoorde und Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, ISBN 0-201-73484-2
- David Abrahams und Aleksey Gurtovoy, *C++ Template Metaprogramming*, ISBN 0-321-22725-5
- David R. Musser und Atul Saini, *STL Tutorial and Reference Guide*, ISBN 0-201-63398-1
- Scott Meyers, *Effective C++*, ISBN 0-201-92488-9
- Scott Meyers, *More Effective C++*, ISBN 0-201-63371-X
- Scott Meyers, *Effective STL*, ISBN 0-201-74962-9
- Scott Meyers, *Effective Modern C++*, ISBN 978-1-491-90399-5

Folgende Literatur ist etwas älter, war jedoch wegweisend:

- Bertrand Meyer, *Object-Oriented Software Construction*, Second Edition, 1997
- Grady Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, 1994, ISBN 0-8053-5340-2
- Erich Gamma et al, *Design Patterns*, ISBN 0-201-63361-2
- Booch, Jacobson, and Rumbaugh, *The Unified Modeling Language User Guide*, Addison Wesley, 1999, ISBN 0-201-57168-4 (Referenz zu UML, jedoch nicht sehr gelungen)

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechzeit ist am Mittwoch 10-12 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln!

- Ich helfe auch gerne bei Problemen bei der Lösung von Übungsaufgaben. Bevor Sie völlig verzweifeln, sollten Sie mir Ihren aktuellen Stand per E-Mail zukommen lassen. Dann werde ich versuchen, Ihnen zu helfen.
- Das kann auch am Wochenende funktionieren.

Objekt-orientierte Techniken sind auf dem Wege neuer Programmiersprachen eingeführt worden, um Probleme mit traditionellen Programmiersprachen zu lösen:

- Simula (1973) von Nygaard und Dahl:
  - Erste OO-Programmiersprache.
  - Die Technik wurde eingeführt, um die Modellierung von Simulationen zu erleichtern.

- Smalltalk wurde in den späten 70er-Jahren bei Xeroc PARC entwickelt und 1983 von Adele Goldberg publiziert:
  - Erste radikale OO-Programmiersprache: Alles sind Objekte einschließlich der Klassen.
  - Die Sprache wurde entwickelt, um die Modellierung und Implementierung der ersten graphischen Benutzeroberfläche zu unterstützen.
  - Charakteristisch ist, dass das gesamte Typsystem dynamisch ist und somit keine statische Typsicherheit existiert: *message not understood*

- C++, das sich zu Beginn noch *C with Classes* nannte, begann seine Entwicklung 1979 und gehört damit zu den frühesten OO-Programmiersprachen.
- Bjarne Stroustrup scheiterte in seinem Bemühen, Simulationen mit den zur Verfügung stehenden Lösungen umzusetzen:
  - Simula: (schöne Programmiersprache; unzumutbare Performance)
  - BCPL: (unzumutbare Programmiersprache; hervorragende Performance)
- Entsprechend war das Ziel von Stroustrup, die Effizienz von C mit der Eleganz von Simula zu kombinieren.

Assembler und viele traditionelle Programmiersprachen (wie etwa Fortran, PL/1 und C) bieten folgende Struktur:

- Eine beliebige Zahl von Übersetzungseinheiten, die unabhängig voneinander zu sogenannten Objekten übersetzt werden können, lassen sich durch den Binder zu einem ausführbaren Programm zusammenbauen.
- Jede Übersetzungseinheit besteht aus global benutzbaren Funktionen und Variablen.
- Parameter und globale Variablen (einschließlich den dynamisch belegten Speicherflächen) werden für eine mehr oder weniger unbeschränkte Kommunikation zwischen den Übersetzungseinheiten verwendet.

- Anwendungen in traditionellen Programmiersprachen tendieren dazu, sich rund um eine Kollektion globaler Variablen zu entwickeln, die von jeder Übersetzungseinheit benutzt und modifiziert werden.
- Dies erschwert das Nachvollziehen von Problemen (wer hat den Inhalt dieser Variable verändert?) und Änderungen der globalen Datenstrukturen sind nicht praktikabel.

Nachfolger der traditionellen Programmiersprachen (wie etwa Modula-2 und Ada) führten Module ein:

- Module schränken den Zugriff ein, d.h. es sind nicht mehr alle Variablen und Prozeduren global zugänglich.
- Stattdessen wird eine Schnittstelle spezifiziert, die alle öffentlich nutzbaren Prozeduren und Variablen aufzählt.
- Abstrakte Datentypen erlauben den Umgang mit Objekten, deren Innenleben verborgen bleibt.
- Dies erlaubt das Verbergen der Datenstrukturen hinter Zugriffsprozeduren.

- Die abstrakten Schnittstellen sind nicht wirklich getrennt von den zugehörigen Implementierungen, d.h. zwischen beiden liegt eine 1:1-Beziehung vor (zumindest aus der Sicht eines zusammengebauten Programms).
- Entsprechend können nicht mehrere Implementierungen eine Schnittstelle gemeinsam verwenden.

- Alle Daten werden in Form von Objekten organisiert (mit Ausnahme einiger elementarer Typen wie etwa dem für ganze Zahlen).
- Auf Objekte wird (explizit oder implizit) über Zeiger zugegriffen.
- Objekte bestehen aus einer Sammlung von Feldern, die entweder einen elementaren Typ haben oder eine Referenz zu einem anderen Objekt sind.
- Objekte sind verpackt: Ein externer Zugriff ist nur über Zugriffsprozeduren möglich (oder explizit öffentliche Felder).

- Eine Klasse assoziiert Prozeduren (Methoden genannt) mit einem Objekt-Typ. Im Falle abstrakter Klassen können die Implementierungen der Prozeduren auch weggelassen werden, so dass nur die Schnittstelle verbleibt.
- Der Typ eines Objekts (der weitgehend in den OO-Sprachen durch eine Klasse repräsentiert wird) spezifiziert die externe Schnittstelle.
- Objekt-Typen können erweitert werden, ohne die Kompatibilität zu ihren Basistypen zu verlieren. (In Verbindung mit Klassen wird hier gelegentlich von Vererbung gesprochen.)
- Objekte werden von einer Klasse mit Hilfe von Konstrukturen erzeugt (instantiiert).

Es gibt eine Vielzahl von OO-Sprachen, die mit sehr unterschiedlichen Ansätzen in folgenden Bereichen arbeiten:

- Die Verpackung (d.h. die Eingrenzung der Sichtbarkeit) kann über Module, Klassen, Objekten oder über spezielle Deklarationen wie etwa den *friends* in C++ erfolgen.
- Die Beziehungen zwischen Modulen, Klassen und Typen werden unterschiedlich definiert.
- Die Art der Vererbung bzw. Erweiterung: einfache vs. mehrfache Vererbung bzw. Erweiterung von Typen vs. Erweiterung von Klassen.
- Wie wird im Falle eines Methoden-Aufrufs bei einem Objekt der zugehörige Programmtext lokalisiert? Das ist nicht trivial im Falle mehrfacher Vererbung oder gar Multimethoden.
- Wann findet die Lokalisierung statt? Nur zur Laufzeit oder auch teilweise zur Übersetzzeit?

- Aufrufketten durch Erweiterungshierarchien (von der abgeleiteten Klasse hin zur Basisklasse oder umgekehrt).
- Statische vs. dynamische Typen.
- Automatische Speicherbereinigung (*garbage collection*) vs. explizite manuelle Speicherverwaltung.
- Organisation der Namensräume.
- Unterstützung für Ausnahmenbehandlungen und generische Programmierung.
- Zusätzliche Unterstützung für aktive Objekte, Aufruf von Objekten über das Netzwerk und Persistenz.
- Unterstützung traditioneller Programmiertechniken.

- Generische Module sind eine Erweiterung des Modulkonzepts, bei der Module mit Typen parametrisiert werden können.
- Ziel der generischen Programmierung ist die Erleichterung der Wiederverwendung von Programmtext, was insbesondere bei einer 1:1-Kopplung von Schnittstellen und Implementierungen ein Problem darstellt.
- Generische Module wurden zunächst bei CLU eingeführt (Ende der 70er Jahre am MIT) und wurden dann insbesondere bekannt durch Ada, das sich hier weitgehend an CLU orientierte.

- Traditionelle OO-Techniken und generische Module sind parallel entwickelte Techniken zur Lösung der Beschränkungen des einfachen Modulkonzepts.
- Beides sind völlig orthogonale Ansätze, d.h. sie können beide gleichzeitig in eine Programmiersprache integriert werden.
- Dies geschah zunächst für Eiffel (Mitte der 80er Jahre) und wurde später bei Modula-3 und C++ eingeführt.
- OO-Techniken können prinzipiell generische Module ersetzen, umgekehrt ist das jedoch schwieriger.
- Beide Techniken haben ihre Stärken und Schwächen:
  - OO-Techniken: Erhöhter Aufwand zur Lokalisierung des Programmtexts und mehr Typüberprüfungen zur Laufzeit; flexibler in Bezug auf dynamisch nachgeladenen Modulen
  - Generische Module: Potential für eine höhere Laufzeiteffizienz, jedoch inflexibel gegenüber dynamisch nachgeladenen Modulen

- Die Metaprogrammierung erlaubt es, den Übersetzer selbst zu programmieren.
- Obwohl eine Metaprogrammierung für C++ ursprünglich nicht vorgesehen worden ist, wurde sie durch spezielle Template-Techniken möglich. Aus heutiger Sicht gehört die Metaprogrammierung zu den wesentlichen Elementen von C++ – auch wenn dies häufig wenig sichtbar in den Bibliotheken verborgen ist.
- Durch den Ausbau der zur Übersetzzeit ausgewerteten **constexpr**-Funktionen (eingeführt in C++11, erheblich erweitert für C++14) hat sich die Metaprogrammierung vereinfacht.
- Auch durch spezielle Bibliotheken für die Metaprogrammierung wird die Anwendung vereinfacht.
- Der Vorteil der Metaprogrammierung liegt darin, mehr Dinge bereits zur Übersetzzeit zu bestimmen, so dass dies nicht mehr zur Laufzeit geschehen muss mit dem wesentlichen Punkt, dass alles zur Übersetzzeit überprüft werden kann.