Programmierkurs

F. Spanier

Institut für Theoretische Astrophysik

Blockkurs Sommer 2022

Original Authors: Dr. Ole Klein, Dr. Steffen Müthing

Administratives

Vorlesung

- 31. März - 13. April
- Montag - Freitag
- jeweils 09:15 -12:00

Tutorium

- Montag - Freitag
- jeweils 14:00 - 17:00
- Übungsaufgaben werden ausschließlich im Tutorium bearbeitet
- Lösungen werden mit Tutoren besprochen

Informationen, Übungsblätter etc.

https://moodle.uni-heidelberg.de/course/view.php?id=11269

Bei Fragen

https://moodle.uni-heidelberg.de/mod/forum/view.php?id=587144

Übungen

- Um Programmieren zu lernen, muss man programmieren!
- Übungszettel enthalten ausschließlich Programmieraufgaben
- Jeden Tag ein neuer Übungszettel
- Übungen werden vor Ort im Tutorium gerechnet
- Abwicklung über MUESLI und Moodle
  - `https://muesli.mathi.uni-heidelberg.de/lecture/view/1487`

Klausur

- Termin: 28.04.2022
- Ort und Zeit unbestimmt

Modul — Lernziele und Inhalt

Lernziele:

Die Studierenden können selbstständig Programme und Lösungen von Programmieraufgaben in C++ entwerfen, realisieren und testen[, und] sind in der Lage mit gängigen Programmierwerkzeugen und Tools unter Linux umzugehen.

Inhalt:

Die Lehrveranstaltung vertieft die Programmierkenntnisse aus dem Modul Einführung in die Praktische Informatik (IPI). Im Vordergrund steht der Erwerb praktischer Fähigkeiten. Die Studierenden lernen algorithmische Lösungen systematisch in Programme umzusetzen. Es wird die Programmiersprache C++ unter dem Betriebssystem Linux verwendet. [. . .]

Modul
Voraussetzungen und Prüfungsmodalitäten

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Voraussetzungen: keine (auch nicht IPI!)

Prüfungsmodalitäten:

Erfolgreiche Teilnahme an den Gruppenübungen und erfolgreiche Teilnahme an einer schriftlichen Prüfung. Im Wintersemester wird am Ende der Vorlesungszeit eine Klausur angeboten. Wird diese nicht bestanden so kann die Prüfungsleistung in einer zweiten Klausur vor Beginn der nächsten Vorlesungszeit erbracht werden. Im Sommersemester wird nur eine Klausur angeboten.

	Programming	Demon Summoning
Must know language unspoken by mankind		
Requires that you be exact or suffer dire consequences		
Involves much cursing, swearing of oaths, and pleading with a higher power		
Not understanding the the true power you wield or the consequences of your actions		
Sometimes you have to execute a child		
Candles		

Das eigentliche Ziel. . .

. . . ist, dass Sie nach diesem Kurs endlich schwerwiegende Fehler beim Programmieren machen!

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Wer versteht folgendes Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Wer versteht folgendes Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

- Wer versteht folgenden C++-Code?

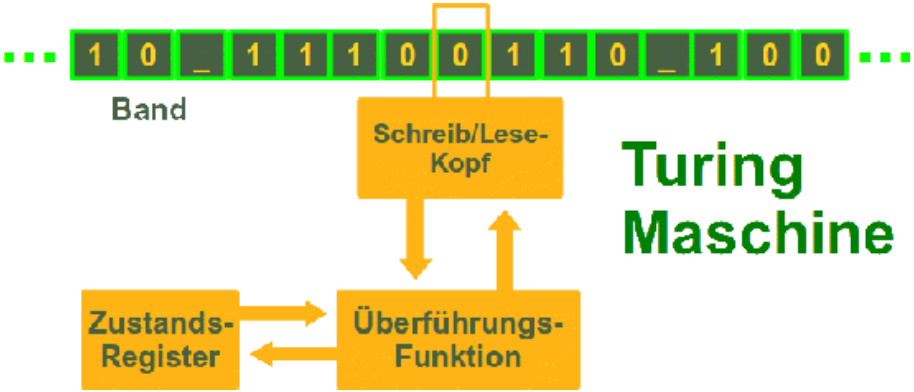
```
auto v = std::array<int>{1,2,3,4};
for (auto& x : v)
  x *= x;
std::cout << std::accumulate(v.begin(),v.end(),0)
          << std::endl;
```

Turing-Vollständigkeit

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Eine Turing-Maschine ist eine einfache Maschine.
- Sie besteht aus:
  - einem potentiell unendlichen Band, welches in Felder eingeteilt ist und pro Feld genau ein Zeichen aufnehmen kann,
  - einem Schreib-Lesekopf sowie
  - einem internen Zustand.
- Je nach Zustand und Inhalt des Bandes kann die Turing-Maschine folgende Aktionen ausführen:
  - ein neues Zeichen Schreiben,
  - den Schreib-Lesekopf um eine Position nach rechts oder links bewegen und
  - den internen Zustand verändern.

Turing-Vollständigkeit



Turing-Vollständigkeit

- Alles, was berechenbar ist, kann durch Turing-Maschinen berechnet werden
- Eine Programmiersprache, in der man eine Turingmaschine schreiben kann, ist genauso mächtig wie die Turing-Maschine
- Da Turing-Maschinen aber bereits das charakterisieren, was berechenbar ist, handelt es sich bei der Sprache um eine berechnungs-universelle Sprache, d.h. wir können damit genau die berechenbaren Funktionen programmieren.

Programmierparadigma

„Ein Programmierparadigma ist ein fundamentaler Programmierstil.“

- Darstellung von statischen und dynamischen Elementen
- Struktureller Aufbau eines Programms
- Wichtig: Imperativ vs. deklarativ

Programmierparadigma

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Imperativ

- Programme sind Folge von Befehlen
- Untermenge: Strukturierte Progrmmierung („*Go To Statement Considered Harmful*“)
- Ebenfalls Untermenge: Prozedurale Programmierung (Unterteilung von Programmen in Unterprogramme/Prozeduren)
- Noch eine Untermenge: Modulare Programmierung (Module, z.B. Bibliotheken, gliedern große Softwareprojekte)

Programmierparadigma

Deklarativ

- Problem wird beschrieben, Lösungsweg wird „automatisch“ ermittelt
- Beispiel: Funktionale Programmierung (alles ist eine Funktion)
- Beispiel: Logische Programmierung (Axiome definieren Regeln)
- Fun fact: Mit dem *make* Tool lernen wir hier eine deklarative Sprache kennen

Programmierparadigma

Generisch

- Allgemeiner Entwurf von Funktionen
- Verwendbar für verschiedene Datentypen

Programmierparadigma

Objekt-Orientiert

1. Alles ist ein Objekt
2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen)
3. Objekte haben ihren eigenen Speicher (strukturiert als Objekte)
4. Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss)
5. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste)
6. Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt“

Alan Kay

Wieso nicht. . .

Ook!

Esoterische Sprache für Orang-Utans, die Turing-vollständig ist

Befehl	Beschreibung
Ook. Ook.	den Wert der aktuellen Zelle um 1 erhöhen
Ook! Ook!	den Wert der aktuellen Zelle um 1 verringern
Ook. Ook?	eine Zelle nach rechts gehen
Ook? Ook.	eine Zelle nach links gehen
Ook! Ook?	Schleifenanfang – die Schleife durchlaufen solange der Wert der aktuellen Zelle ungleich 0 ist
Ook? Ook!	Schleifenende – beendet die Schleife, wenn der Wert der aktuellen Zelle gleich 0 ist
Ook! Ook.	den Wert der aktuellen Zelle ausdrucken
Ook. Ook!	einen Wert von der Tastatur in die aktuelle Zelle einlesen

Wieso nicht. . .

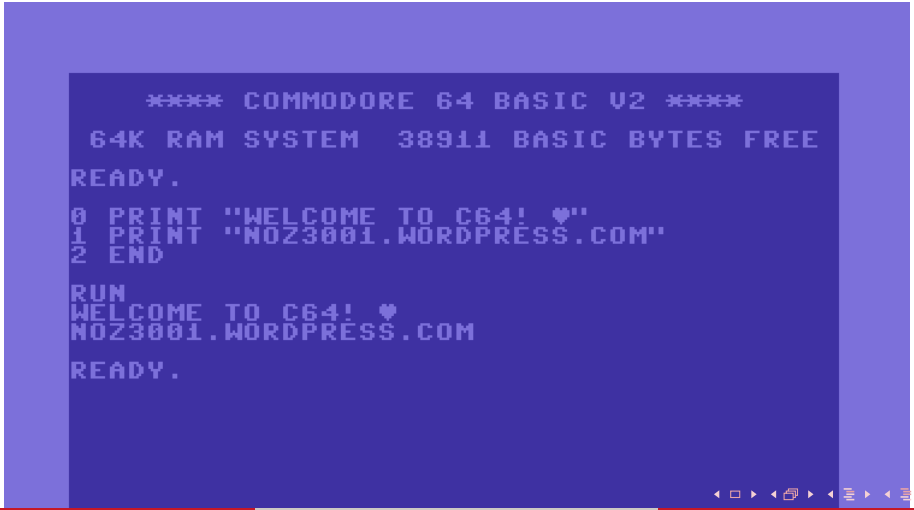
UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Hello World in Ook!

```
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook?
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook! Ook! Ook? Ook!
Ook? Ook. Ook! Ook. Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook?
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook. Ook. Ook. Ook! Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook!
Ook. Ook. Ook? Ook. Ook? Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook! Ook? Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook. Ook! Ook. Ook.
Ook? Ook. Ook? Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook! Ook? Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook. Ook? Ook. Ook? Ook. Ook? Ook.
Ook? Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook! Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook. Ook. Ook?
Ook. Ook? Ook. Ook. Ook! Ook.
```

Wieso nicht. . .

C64-Basic



Wieso nicht. . .

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

C64-Basic

- Triviale Syntax
- Nicht strukturiert, nicht funktional (Spaghetti-Code!)
- Nicht standardisiert!
- Interpreter: Langsam

(Und in Bezug auf den C64: Mit Basic alleine kann eigentlich nichts machen)

Wieso nicht Assembler?


```
global _start

section .text

_start:
  mov rax, 1          ; write(
  mov rdi, 1          ;   STDOUT_FILENO,
  mov rsi, msg        ;   "Hello, world!\n",
  mov rdx, msglen     ;   sizeof("Hello, world!\n")
  syscall             ; );

  mov rax, 60         ; exit(
  mov rdi, 0          ;   EXIT_SUCCESS
  syscall             ; );

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ  - msg
```

Wieso nicht Assembler?

- Nicht strukturiert, nicht objekt-orientiert
- Maschinen-spezifisch $\rightarrow$ nicht portabel
- Quasi undurchsichtig...

Wieso nicht Pascal?

- Enthält wichtige Paradigmen: Objekt-Orientierung, Struktur, Module...
- Nicht überall verfügbar
- Nicht vollständig standardisiert

Wieso nicht Fortran?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Enthält wichtige Paradigmen: Objekt-Orientierung, Struktur, Module...
- Überall verfügbar
- Gut standardisiert
- Bis ins 21. Jahrhundert wichtige Sprache für naturwissenschaftliche Anwendungen
- Objekt-Orientierung wesentlich schlechter als in C++

Wieso nicht Python?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Wieso nicht Python?

- Enthält wichtige Paradigmen: Objekt-Orientierung, Struktur, Module...
- Überall verfügbar
- Gut standardisiert
- Meistens langsam...
- ...außer man verwendet in C++ geschriebene Bibliotheken

Wieso nicht C#?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
    static void Main(string[] args)
    {
    }
    }
}
```

Wieso nicht C#?

- Weiterentwicklung von C++
- Nicht überall verfügbar
- Genauso fürchterlich wie Java

Was ist C++?

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

C++ ist eine Programmiersprache, die sowohl für systemnahe als auch für Anwendungsprogrammierung entwickelt wurde.

- Unterstützung verschiedener Programmier-Paradigmen: imperativ, objekt-orientiert, generisch
- Fokus auf Effizienz, Performance und Flexibilität
- Einsatzgebiete von embedded Controllern bis zu Supercomputern
- Erlaubt direkte Verwaltung von Hardware-Ressourcen
- “Zero-cost abstractions”
- “Pay only for what you use”
- Offener Standard mit mehreren Implementierungen
- Wichtigste Compiler:
  - Open Source: GCC und Clang (LLVM)
  - Kommerziell: Microsoft und Intel

Geschichte

1979 Entwicklung “C with Classes” durch Bjarne Stroustrup
1985 Erster kommerzieller C++-Compiler
1989 C++ 2.0
1998 Standardisierung als ISO/IEC 14882:1998 (C++98)
2011 Nächste große Version mit neuen Funktionen (C++11)
2014 C++14 mit vielen kleinen Fehlerkorrekturen und praktischen Features
2017 C++17 weitere Language Features und Library Erweiterungen
2020 C++20, aktuelle Version von C++ (noch nicht besonders verbreitet)
2023 C++23, kommende Version, wird erst in den nächsten Jahren Verwendung finden

Warum UNIX / Linux?

Die meisten Arbeitsgruppen in Mathematik, Physik und Informatik verwenden zumindest in Teilen Linux. Daher werden sich die meisten von Ihnen früher oder später damit auseinandersetzen müssen.

Linux ist mit Abstand das am häufigsten genutzte Unix. Über mit Linux betriebene Webserver werden nicht nur große Teile des Internet zur Verfügung gestellt, grob 95% der 500 schnellsten Rechner der Welt basieren auf Linux, und alle unter den zehn schnellsten. Wer mit Rechnern wissenschaftlich arbeiten will, kommt an Unix nicht vorbei.

Darüber hinaus ist Linux durch seine offene Natur und die vielen Open-Source-Komponenten eine hervorragende Spielwiese für angehende Informatiker. Für jedes Programm, das auf einem normalen Linux-Desktop installiert ist, können Sie den Quellcode herunterladen und lernen, wie es programmiert wurde!

Was ist UNIX?

Im engeren Sinne Unix (1969), ein Betriebssystem, das viele Funktionen einführte, die in heutigen Betriebssystemen selbstverständlich sind.
Im weiteren Sinne jedes Betriebssystem, das sich an die UNIX-Interfaces und -Spezifikation hält:

- Free,Net,OpenBSD, basierend auf einer Unix-Variante von der UC Berkeley
- macOS, das auf Teilen von FreeBSD und NetBSD basiert
- iOS, aus macOS hervorgegangen
- illumos/OpenIndiana, basierend auf Solaris und System V
- Linux (1991), streng genommen kein UNIX, aber weitestgehend kompatibel
- Android, von Google stark modifiziertes Linux
- Viele embedded systems, oft auf Basis von BSD oder Linux, in Netzwerkroutern, Fernsehern, Robotern, Autos, Raketen, ...

Linux-Installation

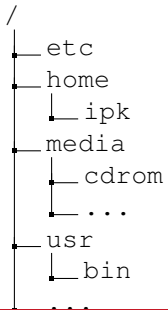
- Fester Teil des Moduls ist die Entwicklung von Kompetenz im Umgang mit Linux
- Dazu benötigen Sie natürlich Zugang zu einem solchen System
- Möglichkeiten für Sie:
  - Linux/macOS evtl. bereits vorhanden
  - Für Interessierte/Fortgeschrittene: Linux parallel zu Windows installieren
  - Für Nutzer von Windows 10: Windows Subsystem for Linux (WSL) installieren
  - Ansonsten: Installation einer virtuellen Maschine (VM)

Installationsanleitungen

Anleitungen (WSL und VM) sind auf der Moodle zu finden

Dateisystem I

- Dateien liegen in Verzeichnissen
- Verzeichnistrenner unter Unix:
  `home/user` statt `C:\Users\Name`
- Groß- und Kleinschreibung unter Linux relevant:
  `test.txt` $\neq$ `Test.txt`
- Linux kennt keine Laufwerksbuchstaben, alle Verzeichnisse haben eine gemeinsame Wurzel `/`



Dateisystem II

- Jeder Benutzer hat ein home directory für eigene Dateien, normalerweise in `/home/<username>`
- Normale Benutzer haben keine Schreibrechte außerhalb ihres home directories
- Der Administrator (heißt unter UNIX root) darf überall lesen und schreiben
- Jedes Programm hat ein Arbeitsverzeichnis (working directory)
- Dateizugriffe immer relativ zu diesem Verzeichnis
- Arbeitsverzeichnis kann gewechselt werden
- Spezielle Verzeichnis-Namen
  - `.` Das aktuelle Verzeichnis
  - `..` Das übergeordnete Verzeichnis
  - `/` Das Wurzel-Verzeichnis
  - `~` Das Home Directory des aktuellen Benutzers (funktioniert nicht in jedem Kontext)

Shell I

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Linux verfügt über eine graphische Oberfläche mit der es sich ähnlich bedienen lässt wie die bekannten Desktops von Windows und macOS.
- Es gibt eine Vielzahl an Benutzeroberflächen. Auch welche die Windows nachahmen, macOS nachahmen oder gänzlich neue Wege gehen.
- Für Aufgaben wie Programmieren ist es jedoch weiterhin sinnvoll, sich mit dem schwarz-weißen Terminal-Fenster und der darin laufenden Shell zu befassen:
  - Automatisierung von monotonen und repetitiven Arbeiten
  - Mit etwas Übung ist man bei vielen Aufgaben schneller als in der graphischen Oberfläche (GUI)
  - Viele UNIX-Programme haben keine GUI und können direkt nur über die Shell aufgerufen werden, z.B. der C++-Compiler, den wir zum programmieren in diesem Kurs brauchen

Shell II

Möglichkeiten für erste Schritte mit der Shell:

- Linux / macOS, falls vorhanden (klar!)
- die Windows PowerShell (eingeschränkte Kompatibilität)
- ein Shell-Emulator online

Online Shells

bellard.org Virtuelles Fedora 29 (Fabrice Bellard)

https://bellard.org/jslinux/vm.html?cpu=riscv64&url=fedora29-riscv-2.cfg&mem=256

copy.sh Virtuelles ArchLinux (Fabian Hemmer)

https://copy.sh/v86/?profile=archlinux

Shell III

```
[ipk@vm ~] _
```

- Wichtige Informationen am Anfang der Zeile (prompt)
  - Benutzername
  - Rechnername
  - aktuelles Verzeichnis
- Ausgabe des vollen Pfades mit `pwd`:

```
[ipk@vm ~] pwd
/home/ipk/
```

- Shell beenden mit `exit`:

```
[ipk@vm ~] exit
<Fenster schließt sich>
```

Kommandos in der Shell

```
[ipk@vm ~] cmd -sv --opt --op2 arg1 arg2 ...
```

- Die meisten Kommandos haben ein einheitliches Interface
  - Am Anfang steht der Name des Kommandos (der Dateiname des Programms)
  - Danach folgen Optionen (beginnen mit `"-"`)
  - Am Ende stehen die Argumente ohne `"-"`
- Argumente bestimmen, worauf das Kommando angewendet wird
- Optionen verändern, wie das Kommando arbeitet. Wichtige Optionen haben oft einen langen Namen und eine Abkürzung aus einem Buchstaben
  - Lange Optionen beginnen mit `"--"`
  - Kurze Optionen beginnen mit `"-"` und können gruppiert werden, z.B. `"-rf"`

Hilfe zu Kommandos

- Die meisten Kommandos geben eine kurze Übersicht der erlaubten Optionen und Argumente aus, wenn man sie falsch benutzt:

```
[ipk@vm ~] rm
usage: rm [-f | -i] [-dPRrvW] file ...
       unlink file
```

- Fast alle Kommandos geben mit der Option `"--help"` einen Hilfetext aufs Terminal aus
- Für genauere Informationen gibt es die man pages, die man mit dem Befehl `man` aufruft

```
[ipk@vm ~] man gcc
```

- In der man page bewegt man sich mit den Pfeiltasten und verlässt sie mit der Taste `"q"`

Verzeichnis wechseln I

- Verzeichnis wechseln mit `cd <name>` (change directory):

```
[ipk@vm ~] cd Documents
[ipk@vm Documents]
```

- Neues Verzeichnis wird im prompt angezeigt
- `cd` erzeugt UNIX-typisch nur bei Fehlern eine Ausgabe

```
[ipk@vm Documents] cd nonesuch
-bash: cd: nonesuch: No such file or directory
[ipk@vm Documents]
```

- `cd ..` kehrt ins übergeordnete Verzeichnis zurück

```
[ipk@vm Documents] cd ..
[ipk@vm ~]
```

Verzeichnis wechseln II

- `cd` unterstützt auch zusammengesetzte Pfade

```
[ipk@vm ~] cd Documents/c++
[ipk@vm c++]
```

- Man kann auch absolute Pfade verwenden, die mit `/` beginnen und unabhängig vom aktuellen Verzeichnis sind:

```
[ipk@vm c++] cd /etc/sysconfig
[ipk@vm sysconfig]
```

- `cd` ohne Argumente wechselt immer ins home directory

```
[ipk@vm sysconfig] cd
[ipk@vm ~]
```

Dateien auflisten

- `ls` (list) zeigt die Dateien im aktuellen Verzeichnis an

```
[ipk@vm c++] ls
helloworld    helloworld.cc
```

- Dateien, die mit einem Punkt beginnen, werden normalerweise nicht angezeigt. Dies kann man mit der Option `-a` ändern:

```
[ipk@vm c++] ls -a
.             ..            .versteckt
helloworld    helloworld.cc
```

- Auch `ls` akzeptiert einen Pfad als Argument

```
[ipk@vm c++] ls ~/Documents
c++
```

- `ls -l` (long) zeigt zusätzliche Informationen wie Besitzer, Zugriffsrechte, Dateigröße etc.

Dateien kopieren und verschieben

- `cp` (copy) kopiert, `mv` (move) verschiebt Dateien

```
[ipk@vm c++] cp original copy
[ipk@vm c++] ls
copy     original
```

- Man kann auch mehrere Dateien gleichzeitig kopieren. In diesem Fall muss das Ziel ein Verzeichnis sein

```
[ipk@vm c++] mkdir subdir
[ipk@vm c++] mv original copy subdir
[ipk@vm c++] ls
subdir
[ipk@vm c++] ls subdir
copy     original
```

- Mit der Option `-r` (recursive) kopiert man Unterverzeichnisse samt Inhalt, beim

Dateien löschen

- `rm` (remove) löscht Dateien und Verzeichnisse

```
[ipk@vm c++] rm subdir/copy
[ipk@vm c++] ls subdir
original
```

- Mit der Option `-r` (recursive) löscht `rm` ein Verzeichnis mit allen Inhalten

```
[ipk@vm c++] rm -r subdir
[ipk@vm c++] ls
```

Warnung

`rm` fragt nicht nach, bevor die Dateien gelöscht werden, und in der Shell gibt es keinen Papierkorb! Gelöschte Dateien können nicht wiederhergestellt werden!

Dateien bearbeiten

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Es gibt Editoren, die im Textmodus im Terminal arbeiten (`vim`, `nano`, `emacs`,...), aber wir werden in dieser Vorlesung Editoren mit GUI verwenden
- Im Pool sind die Editoren `gedit` und `kate` installiert, in der VM `geany` und `qtcreator`, bei der WSL-Anleitung werden Sie `geany` installieren
- Sie können den Editor entweder wie gewohnt per Doppelklick auf eine Datei im Dateimanager starten oder direkt über die Shell

```
[ipk@vm c++] geany helloworld.cc &
[ipk@vm c++]
```

- Beim Starten von GUI-Programmen in der Shell ist es sinnvoll, ein `"&"` ans Ende des Befehls zu setzen. Ansonsten ist die Shell blockiert, bis Sie das gestartete Programm wieder beenden.

Dateien anzeigen

- `cat` zeigt den Inhalt von Dateien im Terminal an

```
[ipk@vm c++] ls
file1 file2
[ipk@vm c++] cat file2 file1
line in file2
line in file1
another line in file1
```

- `cat` kann bei langen Dateien unübersichtlich werden. `less` öffnet die Dateien in einem Viewer ähnlich zu dem für die man pages

```
[ipk@vm c++] less file1
```

  - Navigieren mit Pfeiltasten und `"q"` zum Beenden
  - Leertaste, um einen Bildschirm weiter zu springen
  - `"/"` + Suchwort + Enter, um zu suchen
  - `"n"`, um zum nächsten Vorkommen des Suchworts zu springen

Dateien durchsuchen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- `grep <pattern> <datei>...` durchsucht Dateien nach einem Pattern

```
[ipk@vm c++] grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
10:operator:x:11:0:operator:/root:/sbin/nologin
```

- Das Pattern kann ein einfaches Wort sein, man kann aber auch nach komplizierten Ausdrücken mit Hilfe sogenannter Regular Expressions suchen

I/O Streams

- UNIX-Programme kommunizieren mit dem System über sogenannte I/O (Input/Output) streams
- Streams sind eine Einbahnstraße — man kann aus ihnen entweder lesen oder Daten in sie schreiben
- Beim Start hat jedes Programm drei offene Streams

stdin	Die Standardeingabe liest User-Eingaben von der Konsole, ist verbunden mit file descriptor 0
stdout	An die Standardausgabe werden normale Ergebnisse des Programms ausgegeben, ist verbunden mit file descriptor 1
stderr	An die Standardfehlerausgabe werden diagnostische Meldungen wie Fehler ausgegeben, ist verbunden mit file descriptor 2

Umleiten von I/O Streams I

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Normalerweise sind alle Standardstreams mit dem Terminal verbunden
- Manchmal kann es sinnvoll sein, diese Streams in Dateien umzuleiten
- stdout wird mit `"> datei"` in einer Datei gespeichert

```
[ipk@vm ~] ls > files
[ipk@vm ~] cat files
file1
file2
files
```

Die Ausgabedatei wird angelegt, bevor der Befehl ausgeführt wird, daher taucht sie selbst auf

- Fehlermeldungen werden weiterhin im Terminal angezeigt

```
[ipk@vm ~] ls missingdir > files
ls: missingdir: No such file or directory
[ipk@vm ~] cat files
[ipk@vm ~]
```

Umleiten von I/O Streams II

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- stdin wird mit "< datei" aus einer Datei gelesen

```
[ipk@vm ~] cat # ohne Argument gibt cat stdin nach stdout aus
Eingabe am Terminal^D # (CTRL+D) beendet die Eingabe
Eingabe am Terminal
[ipk@vm ~] cat < files
 file1
 file2
 files
```

- stderr wird mit "2> datei" in einer Datei gespeichert

```
[ipk@vm ~] ls missingdir 2> error
[ipk@vm ~] cat error
ls: missingdir: No such file or directory
```

Umleiten von I/O Streams III

- Die Ausgabe eines Befehls kann direkt in die Eingabe des nächsten Befehls umgeleitet werden mit | (der Pipe-Operator)

```
cat text.txt | grep Hallo
```

Gibt test.txt aus und `grep` sucht dann Hallo

- Sinnvolles Beispiel

```
cat file2.txt | sort  | uniq > list4.txt
```

Die Datei wird ausgegeben, `sort` sortiert sie (alphabetisch), `uniq` entfernt Doppelungen und das Ergebnis landet in list4.txt

Kompilieren von C++-Programmen

- C++-Programme müssen vor dem Ausführen kompiliert (= in Maschinensprache übersetzt) werden
- Der Standard-C++-Compiler unter Linux heißt `g++`
- Oft wird auch gerne stattdessen `clang++` aus dem LLVM-Projekt verwendet, da dieser verständlichere Fehlermeldungen generiert
- Sie sollten immer die Option `-Wall` verwenden, damit der Compiler Sie bei Problemen warnt, die zwar legales C++ sind, aber wahrscheinlich von Ihnen so nicht gewollt sind
- Beim Ausführen des erstellten Programms muss `./` vorangestellt werden

```
[ipk@vm ~] g++ -Wall -o test test.cc
[ipk@vm ~] ./test
```

Ein erstes C++-Programm

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

```cpp
/* make parts of the standard library available */
#include <iostream>
#include <string>

// custom function that takes a string as an argument
void print(std::string msg)
{
  // write to stdout
  std::cout << msg << std::endl;
}

// main function is called at program start
int main(int argc, char** argv)
{
  // variable declaration and initialization
  std::string greeting = "Hello, world!";
  print(greeting);
  return 0;
}
```

Zentrale Bestandteile eines C++-Programms

- Include-Direktiven, um Bibliotheken zu verwenden
  - Stehen am Anfang des Programms
  - Werden in Zukunft erwähnt, wenn erforderlich
  - Nur eine Include-Direktive pro Zeile
- Eigene Funktionen
  - Ähnlich wie mathematische Funktionen mit Parametern und Rückgabewert
  - Jedes Programm muss die Funktion

```
int main(int argc, char** argv)
{
  ...
}
```

enthalten. Diese wird vom Betriebssystem beim Programmstart ausgeführt.

Kommentare

- Kommentare dürfen überall stehen und erklären das Programm für andere Progammierer
- Kommentare mit `//` gehen bis zum Ende der Zeile:

```
int i = 42; // the answer
int x = 0;
```

- Mehrzeilige Kommentare werden durch `/*` begonnen und durch `*/` beendet:

```
/* This comment spans
   multiple lines */
int x = 0;
```

Hinweis

Kommentare erscheinen oft als sinnlose Zusatzarbeit, aber nach einer Woche können Sie sehr dabei helfen, das Programm selber zu verstehen und dem Tutor zu erklären!

Funktionen

- Während der Ausführung eines C++-Programm werden Funktionen aufgerufen, beginnend mit der speziellen Funktion `main(int argc, char** argv)`
- Funktionen können andere Funktionen aufrufen
- Funktionsdefinitionen bestehen aus einer Funktionssignatur und einem Funktionsrumpf (body)

```
return-type functionName(arg-type argName, ...) // signature
{
  // function body
}
```

- Die Funktionssignatur legt fest, wie die Funktion heißt und welche Argumente benötigt werden
- Eine Funktion hat immer einen *return type* (genaueres dazu später). Falls die Funktion nichts zurückgeben soll, verwendet man als return type **`void`**
- Der Funktionsrumpf beschreibt, was die Funktion macht und wird durch geschweifte Klammern eingefasst

Statements

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
int i = 0;
i = i + someFunction();
anotherFunction();
return i;
i = 2; // never executed
```

- Eine C++-Funktion besteht aus einer Reihe von Statements, die nacheinander ausgeführt werden
- Statements werden durch ein Semikolon voneinander getrennt
- Das spezielle Statement **`return`** `something;` verlässt sofort die aktuelle Funktion und gibt `something` als Rückgabewert der Funktion zurück
  - Bei **`void`**-Funktionen kann man den Rückgabewert oder das gesamte **`return`** statement weglassen

Variablen

- Variablen dienen dazu, Werte in Programmen zwischenzuspeichern
- Variablen in C++ haben immer einen festen Typ (ganze Zahl, Kommazahl, Text, . . .)
- Variablennamen bestehen aus Groß- und Kleinbuchstaben, Ziffern und Unterstrichen, dürfen aber nicht mit Ziffern beginnen
- Groß- und Kleinschreibung wird unterschieden!
- Bevor eine Variable benutzt werden kann, muss sie deklariert werden
  - Normale Variablen werden durch ein Statement deklariert:

```
variable-type variableName = initial-value;
```

  - Funktionsparameter werden in der Funktionsdeklaration deklariert:

```
void func(var-type1 param1, var-type2 param2)
```

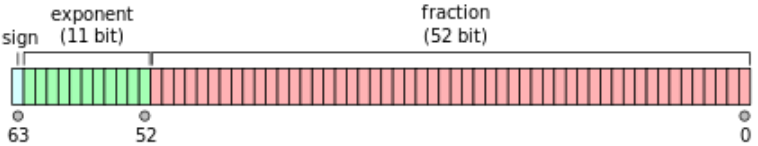
Wichtige Variablen-Typen

C++ kennt viele Typen von Variablen, hier einige wichtige (Zahlenreichweiten gültig auf 64-Bit Linux):

```
// 32-Bit Integer, ganze Zahlen aus [−2^31, 2^31 − 1]
int i = 1;
// 64-Bit Integer, ganze Zahlen aus [−2^63, 2^63 − 1]
long l = 1;
// 8-Bit Integer, ganze Zahlen aus [−2^7, 2^7 − 1]
char c = 1;
// Boolean (Wahrheitswert), true (=1) oder false (=0)
bool b = true;
// Text (Zeichenkette), benötigt #include <string>
std::string msg = "Hello";
// Fließkommazahl doppelter Genauigkeit
double d = 3.141;
// Fließkommazahl einfacher Genauigkeit
float f = 3.141;
```

Integer-Varianten für ausschließlich positive Zahlen durch vorangestelltes **`unsigned`** mit Wertebereich $[0, 2^{\text{bits}} - 1]$

Doppelte Genauigkeit



- Ungefäre Genauigkeit: 16 Dezimalstellen
- Größte Werte: 10^{308}
- Wahl der Fließkommazahl hängt insbesondere von der internen CPU-Breit ab.
- `long integer` sind „genauer“($\rightarrow$ Fixkomma)

Arkanes Wissen: char*

- `char` ist eigentlich ein Zeichen (character) mit einer Länge von 1 Byte (ASCII-Kodierung)
- Früher ™ wurden Zeichenketten als Pointer auf `char` dargestellt (zu Pointern kommen wir später)
- Weil `char` genau ein Byte lang ist, wird ein Pointer auf `char` als Weg genommen, einen Speicherblock von x Byte zu reservieren

Ist das wichtig? Nein.
Wird das so gemacht? Ja!

Scopes und Variablen-Lebenszeit

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Ein block scope ist eine durch geschweifte Klammern eingeschlossene Gruppe von Statements
- Scopes können beliebig geschachtelt werden
- Variablen haben eine begrenzte Lebenszeit:
  - Die Lebenszeit einer Variablen beginnt am Punkt ihrer Deklaration
  - Die Lebenszeit einer Variablen endet, sobald das Programm das scope verlässt, in dem sie deklariert wurde

```
int cube(int x)
{
  // x existiert in der gesamten Funktion
  {
    int y = x * x; // y existiert ab hier
    x = x * y;
  } // y existiert ab hier nicht mehr
  return x;
} // x existiert ab hier nicht mehr
```

Scopes und Namenskollisionen

- Es ist nicht möglich, im selben scope zwei Variablen mit demselben Namen anzulegen

```
{
  int x = 2;
  int x = 3; // Compile-Fehler!
}
```

- Namen in einem inneren Scope überschreiben temporär Namen im äußeren Scope

```
int abs(int x) { ... }

{
  int x = -2;
  {
    double x = 3.3; int abs = -2;
    std::cout << x << std::endl; // gibt 3.3 aus
    x = abs(x); // Compile-Fehler, abs ist eine Variable!
  }
  x = abs(x); // danach: x == 2
}
```

Expressions (Ausdrücke)

Um Dinge in C++ zu berechnen, verwenden wir Expressions (Ausdrücke)

- Ausdrücke sind Kombinationen von Werten, Variablen, Funktionsaufrufen und mathematischen Operatoren, die ein Ergebnis produzieren, das einer Variablen zugewiesen werden kann:

```
i = 2;
j = i * j;
d = std::sqrt(2.0) + j;
```

- Beim Auswerten zusammengesetzter Ausdrücke wie `(a * b + c) * d` gelten Klammern und erweiterte Punkt-Vor-Strich-Regeln, die sogenannte operator precedence

Regelübersicht

`https://en.cppreference.com/w/cpp/language/operator_precedence`

Operatoren für Zahlen-Variablen

- Es gibt die üblichen binären Operatoren `+`,`-`,`*`,`/`
- `a % b` rechnet den Rest der Ganzzahldivision von `a` durch `b` aus:

```
13 % 5 // result: 3
```

- Division rundet bei ganzen Zahlen immer ab
- Bei einer Ganzzahl-Division durch 0 stürzt das Programm ab
- `=` weist seine rechte Seite der linken zu und hat den gleichen Wert wie die Zuweisung

```
a = b = 2 * 21; // both a and b have value 42
```

- Für häufig vorkommende Zuweisungen gibt es Abkürzungen:

```
a += b; // shortcut for a = a + b (also for -,*,/,%)
x = i++; // post-increment, shortcut for x = i; i = i + 1;
x = ++i; // pre-increment,  shortcut for i = i + 1; x = i;
```

- Es gibt auch Pre- und Post-decrement (`--`)

Vergleichsoperatoren

- Vergleichsoperatoren produzieren Wahrheitswerte:

```
4 > 3; // true
```

- Verfügbare Operatoren

```
a < b; // a strictly less than b
a > b; // a strictly greater than b
a <= b; // a less than or equal to b
a >= b; // a greater than or equal to b
a == b; // a equal to b (note the double =)!
a != b; // a not equal to b!
```

Kombination von Wahrheitswerten

Zur Kombination von Testergebnissen gibt es symbolische oder text-basierte Operatoren:

- Kombination mehrerer Tests mit und bzw. oder:

```
a == b ||  b == c; // a equal b or b equal c
a == b or  b == c; // a equal b or b equal c
a == b &&  b == c; // a equal b and b equal c
a == b and b == c; // a equal b and b equal c
```

- Invertierung eines Wahrheitswerts:

```
!true == false;
not true == false;
```

Bitoperatoren

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- AND, OR. . . existieren auch als Bitoperatoren für Zahlen

```
4|3; // 7
4&3; //0
```

- Verfügbare Operatoren

```
a & b; // bitwise AND
a | b; // bitwise OR
a ^ b; // bitwise XOR
a << b; // left shift a by b places
a >> b; // right shift a by b places
~a; // bitwise NOT
```

Der ? Operator

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Eigentlich eine einfache **if** Abfrage

```
expression ? expression : expression;
```

- Folgendes ist äquivalent

```
int a,b,c;
a = (b>c)?0:1;
if (b>c)
        {
                a = 0;
        }
else
        {
                a = 1;
        }
```

Texte / Strings

- Texte bzw. Zeichenketten werden in Variablen vom Typ `std::string` gespeichert und oft als string bezeichnet
- Feste Strings werden im Programm durch doppelte Anführungszeichen eingeschlossen

```
std::string msg = "Hello world!";
```

- Strings können mit + kombiniert werden

```
std::string hello = "Hello, ";
std::string world = "world";
std::string msg = hello + world;
```

- Strings können mit == und != verglichen werden

```
std::string a = "a";
a == "b"; // false
```

Warnung

Beim Vergleich oder Kombinieren von Strings muss links immer eine Variable stehen!

Strings - historisch

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Texte wurden früher (und werden in C) als **`char`**`*` gespeichert

```
char v[4] = "asd"; //valid in C++
```

Es handelt sich um Null-terminierte Zeichenketten (das letzte Zeichen ist `'\0'`)
In vielen Fällen benötigt man diese antiquierte Form statt eines Strings.
Umwandlung durch die `c_str()`-Methode des Strings

Ausgabe auf das Terminal

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Um mit dem Benutzer am Terminal zu kommunizieren, kann unser Programm auf die drei Streams stdin, stdout und stderr zugreifen (vgl. Shell)
- Zur Ausgabe verwenden wir `std::cout`. Alles, was wir ausgeben wollen, “schieben” wir mit `<<` in die Standardausgabe

```
#include <iostream> // required for input / output
...
std::string user = "Joe";
std::cout << "Hello, " << user << std::endl;
```

- Um einen Zeilenumbruch zu erzeugen, gibt man `std::endl` (end line) aus

Eingabe vom Terminal

- Zum Einlesen von Benutzereingaben verwenden wir `std::cin`
- Hierfür muss die Variable vorher deklariert werden
- Eingaben “ziehen” wir mit `>>` aus der Standardeingabe

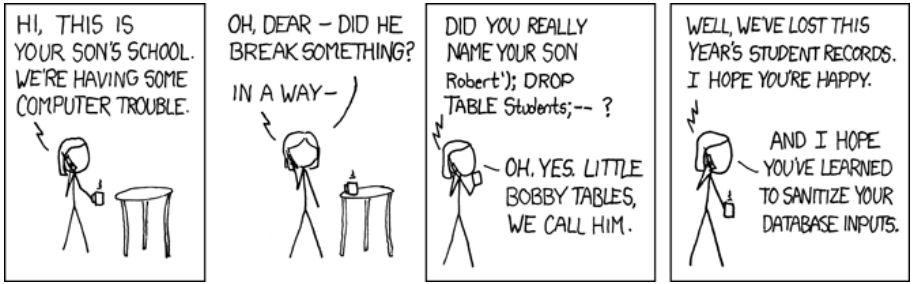
```
#include <iostream>
...
std::string user = "";
int answer = 0;
std::cout << "Enter your name: " << std::endl;
std::cin >> user;
std::cout << "Enter your answer: " << std::endl;
std::cin >> answer;
std::cout << "Hi " << user << "! Your answer was: "
          << answer << std::endl;
```

- Eingaben an der Konsole müssen mit Return abgeschlossen werden

Eingabe vom Terminal

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- User Input kommt direkt aus der Hölle!!!



Quelle: xkcd

Kontrollfluss

Die meisten Programme lassen sich nicht oder nur umständlich als einfache Folge von Statements aufschreiben, die in festgelegter Reihenfolge ausgeführt werden.
Beispiele

- Eine Funktion, die den Betrag einer Zahl zurückgibt
- Eine Funktion, die eine Division durch 0 abfängt und einen Fehler ausgibt
- Eine Funktion, die die Summe aller Zahlen von 1 bis N berechnet
- ...

Programmiersprachen enthalten spezielle Statements, die basierend auf dem Wert einer expression unterschiedliche Anweisungen ausführen

Verzweigungen / Branches

- Das **if**-Statement führt unterschiedlichen Code aus, je nachdem, ob ein Ausdruck wahr oder falsch ist

```
int abs(int x)
{
  if (x > 0)
  {
    return x;
  }
  else
  {
    return -x;
  }
}
```

- Der **else**-Teil der Anweisung ist optional:

```
if (weekday == "Friday")
{
  ipk_lecture();
}
```

Wiederholte Funktionsausführung

- Ein Programm muss oft den gleichen Code wiederholt ausführen, z.B. um

$$\sum_{i=1}^{n} i$$

  zu berechnen
- Zwei Programmieransätze:
  - Rekursion: Die Funktion ruft sich mit veränderten Argumenten selbst wieder auf
  - Iteration: Eine spezielle Anweisung führt einige Statements wiederholt aus

Rekursion

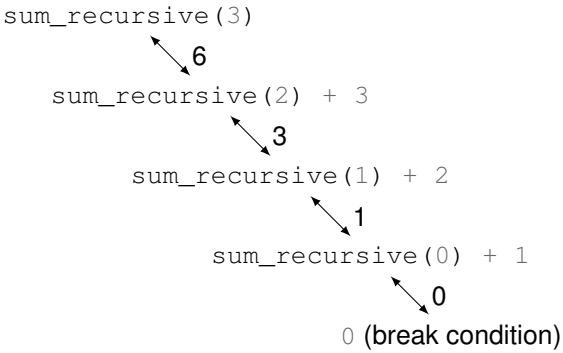
- Idee: Eine Funktion ruft sich mit veränderten Argumenten so oft selbst wieder auf, bis eine Abbruchbedingung erfüllt wird:

```
int sum_recursive(int n)
{
  if (n > 0)
  {
    return sum_recursive(n - 1) + n;
  }
  else
  {
    return 0;
  }
}
```

- Erfordert immer mindestens ein `if`-Statement, bei dem in genau einem Branch die Funktion erneut aufgerufen wird!
- Schlecht geeignet, wenn die Funktion nichts berechnet, sondern nur Seiteneffekte hat (Beispiel: die ersten N Zahlen auf das Terminal ausgeben)

Rekursion: Beispiel

- Berechnung von `sum_recursive(3)`
- Die Zahlen an den Pfeilen geben die Rückgabewerte der Funktionsaufrufe an



Iteration mit while-Schleife

- Eine **while**-Schleife führt den nachfolgenden Block von Statements immer wieder aus, so lange die Bedingung wahr ist

```
int sum_iterative(int n)
{
  int result = 0;
  int i = 0;
  while (i <= n)
  {
    result += i;
    ++i;
  }
  return result;
}
```

- Oft einfacher nachzuvollziehen
- Ist oft etwas expliziter und benötigt mehr Variablen

Endlosschleife

```
while (true)
        {
                //...Anweisungen
        }
```

Wenn man z.B. darauf wartet, dass der Benutzer etwas tut!

Iteration mit for-Schleife I

- Viele Schleifen werden wiederholt für verschiedene Werte einer Zähler-Variablen ausgeführt
- C++ hat eine spezielle **for**-Schleife für solche Fälle:

```
int sum_for(int n)
{
  int result = 0;
  for (int i = 0 ; i <= n ; ++i)
  {
    result += i;
  }
  return result;
}
```

- Sagt dem Leser, dass hier über eine Zählervariable iteriert wird
- Beschränkt die Lebenszeit der Zählervariablen auf die Schleife
- Etwas komplizierter als die **while**-Schleife

Iteration mit for-Schleife II

Jede **for**-Schleife kann in eine äquivalente **while**-Schleife umgewandelt werden

```
for (int i = 0 ; i <= n ; ++i)
{
  ...
}
```

wird zu

```
{
  int i = 0;
  while (i <= n)
  {
    ... ;
    ++i;
  }
}
```

select

```
int i = ...;
switch (i)
{
  case 1:
  case 2:
  case 3:
    do_stuff();
    break;
  case 4:
    do_other_stuff();
    break;
  default:
    do_default();
}
```

Deklaration und Definition von Funktionen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Damit eine Funktion von einer anderen Funktion aus aufgerufen werden kann, muss sie vor Definition der aufrufenden Funktion deklariert worden sein:

```
int error(x)
{
  return square(x); // compile error, place after square
}

int square(int x)
{
  return x * x;
}

int cube(int x)
{
  return square(x) * x; // work just fine
}
```

Deklaration und Definition von Funktionen

- Damit eine Funktion von einer anderen Funktion aus aufgerufen werden kann, muss sie vor Definition der aufrufenden Funktion deklariert worden sein:

```
int square(int x); // Declaration

int error(x)
{
        return square(x); // no error!
}

int square(int x) // Definition
{
        return x * x;
}

int cube(int x)
{
        return square(x) * x;
}
```

Deklaration vs. Definition

- Bevor man eine Funktion in C++ verwenden kann, muss sie deklariert werden.
- Eine Deklaration sagt dem Compiler nur, dass es eine Funktion mit einer bestimmten Signatur gibt.
- Deklarationen sind Funktionsköpfe, bei denen statt Code ein Semikolon folgt:

```
double cube(double x);
```

- Eine Definition enthält den eigentlichen Programmcode, wie bekannt.
- Eine Funktion darf beliebig oft deklariert werden, aber nur einmal definiert (one definition rule).
  - Deklaration → Header (`.hh`-Datei, der Inhalt taucht in jeder `.cc`-Datei auf, die den Header inkludiert).
  - Definition → Implementation (`.cc`-Datei).

Beispiel

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

cube.hh

```
// function for calculating the cube of a double
double cube(double x); // <-- declaration
```

cube.cc

```
#include "cube.hh" // preprocessor: replaced by declaration
double cube(double x) // <-- definition
{
  return x * x * x;
}
```

main.cc

```
#include <iostream>
#include "cube.hh" // also replaced by declaration
int main(int argc, char** argv)
{ // meaning of "cube" is clear thanks to declaration
  std::cout << cube(3.0) << std::endl;
}
```

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (`.cc`-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

// function for calculating the cube of a double
double cube(double x);

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (`.cc`-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

// function for calculating the cube of a double
double cube(double x);

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Hinweis

Alle Header-Dateien, die Sie in diesem Kurs schreiben, müssen einen Header-Guard haben!

Namespaces und Zusammenarbeit

- Alice schreibt eine Bibliothek mit einer Funktion `greeting()`, die Sie wie folgt verwenden können:

```
#include <alice.hh>

int main(int argc, char** argv)
{
  greeting(); // aus alice.hh
}
```

Namespaces und Zusammenarbeit

- Alice schreibt eine Bibliothek mit einer Funktion `greeting()`, die Sie wie folgt verwenden können:

```
#include <alice.hh>

int main(int argc, char** argv)
{
  greeting(); // aus alice.hh
}
```

- Bob schreibt auch eine Bibliothek mit einer Funktion `greeting()`:

```
#include <bob.hh>

int main(int argc, char** argv)
{
  greeting(); // aus bob.hh
}
```

Kombinieren mehrerer Bibliotheken

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Wenn man mehrere verschiedene Bibliotheken verwendet, die beide Symbole mit gleichem Namen definieren, kommt es zu Namenskollisionen:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
  // Compile-Fehler!
  // Welches greeting() soll aufgerufen werden?
  greeting();
}
```

Kombinieren mehrerer Bibliotheken

Wenn man mehrere verschiedene Bibliotheken verwendet, die beide Symbole mit gleichem Namen definieren, kommt es zu Namenskollisionen:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
  // Compile-Fehler!
  // Welches greeting() soll aufgerufen werden?
  greeting();
}
```

Problematisch, man kann ja nicht voraussehen, welche Funktionsnamen andere Entwickler sich ausdenken!

Namespaces

- Namespaces ermöglichen es Bibliotheken, eigene Namensräume zu definieren.
- Um auf ein Symbol `func()` aus dem namespace `mylib` zuzugreifen, stellt man den namespace gefolgt von `::` vor den Namen des Symbols:

```
mylib::func()
```

- Namespaces können auch ineinander verschachtelt werden:

```
mylib::mysublib::func()
```

- C++ liefert eine Standardbibliothek als Teil des Compilers mit. Alle Funktionalität der Standardbibliothek befindet sich in namespace `std`.

Namespaces: Alice und Bob

Mit namespaces läßt sich das Problem des uneindeutigen Funktionsnamens auflösen:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
  alice::greeting();
  bob::greeting();
}
```

Namespaces selber anlegen

- Namespaces sind ein spezielles Scope ausserhalb einer Funktion, das durch das Keyword **namespace** eingeleitet wird:

```
namespace alice {

  void greeting()
  {
    std::cout << "Hello Alice!" << std::endl;
  }

} // end namespace alice
```

- Da ein **namespace**-Scope oft sehr lang ist, sollte man hinter die schliessende Klammer einen kurzen Kommentar schreiben, was hier eigentlich geschlossen wird.
- Innerhalb des **namespace**-Scope muss man für andere Funktionen im gleichen namespace den Namen des namespace nicht davorschreiben.

Pointer I

- Zu jeder Variable gehört eine Speicheradresse
- Die Adresse lässt sich mit dem `&`-Operator herausfinden

```
int a = 25;
cout << &a; //gibt die Adresse von a aus
```

- Für die Adressen gibt es den Pointer-Typ

```
int * b;
b = &a; //b ist ein Pointer auf die Adresse von a
```

Pointer II

- Das Gegenstück zum `&`-Operator ist der Dereferenzierungsoperator `*`

```
int a = 25;
int * b;
b = &a; //b ist ein Pointer auf die Adresse von a
cout <<*b; //ergibt 25
```

- Der Pointer ist nicht die Variable, sondern eben nur ein Verweis darauf!

Pointer III

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Pointer haben keinen zugehörigen Speicherplatz, der muß erst reserviert werden.

```
int * b;      //erstelle Pointer
b = new int; //reserviere Speicher
*b = 25;      //Schreibe in den Speicer einen Wert
delete b;     //gib den Speicher wieder frei
```

- Nach der Freigabe mit **delete** weist der Pointer auf keinen gültigen Speicher
- Weist der Pointer auf NULL (nicht 0!), ist er nicht initialisiert.
- Vergessene Freigaben, Verwendung nicht-initialisierter Pointer hat schreckliche Konsequenzen

Pointer IV

- Mit **new** können Arrays reserviert werden

```
int * b;      //erstelle Pointer
b = new int[512]; //reserviere Speicher
b[12] = 25;      //Schreibe an 13. Position in den Speicher einen Wert
delete b;     //gib den Speicher wieder frei
```

- Alternativ geht das mit

```
int b[5];      //erstelle Array
```

aber nur für zur Kompilierzeit bekannte Feldgrößen!

- Wir lernen später bessere Methoden für Arrays mit der Standardlib kennen

Pointer V - Call by Reference

- Mit Pointern versteht man auch Call-by-Reference

```
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

- Hier wird faktisch nicht ein Wert, sondern die Referenz übergeben
- Bei Aufruf von `swap(a,b)` werden die Werte nicht nur in der Funktion vertauscht.

Konstanten I

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Klassischer Weg: `#define`
- Ein Präprozessor-Makro: Vor dem Kompiliervorgang wird der Text im Quellcode ersetzt

```
##define SYMBOL //z.B. bei Headerguard
        #define PI 3.1415 //Zahlenkonstante
        #define MAX(a, b) (a>b?a:b) //Makro
```

- Wenig sinnvoll, Makros lassen sich durch inline-Funktionen ersetzen
- Unklarer Typ

Konstanten II

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Besserer Weg: **const**
- Schlüsselwort vor der Variablendeklaration

```
// constant_values1.cpp
int main() {
   const int i = 5;
   i = 10;   // error: assignment of read-only variable ‘i’
   i++;            // error: increment of read-only variable ‘i’
}
```

Type conversions I

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Was tut folgender Code?

```
#include <iostream>

int main(int argc, char **argv)
{
        std::cout <<1./2. <<std::endl;
        std::cout <<1/2 <<std::endl;
}
```

Type conversions II

```
fspanier@thomson:~$ ./conv
0.5
0
fspanier@thomson:~$
```

Aber wieso???

- Der Code macht Annahmen darüber, welchen Typ die Zahlenkonstanten haben
- 2: `int`, 2. `float`
- Für 1/2 wird also ein Ganzzahldivision durchgeführt, deren Ergebnis auf eine Ganzzahl abgerundet wird

Type conversions III

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Konvertierung von einem Typ in einen anderen findet bei kompatiblen Typen automatisch statt
- Trivial, wenn der neue Typ „größer“ ist (**int** → **long**, **float** → **double**)
- Problematisch, wenn der Typ „kleiner“ ist

```
#include <iostream>

int main(int argc, char **argv)
{
        long a = 12345678901;
        int b;
        b = a;
        std::cout <<a <<std::endl; //12345678901
        std::cout <<b <<std::endl; //-539222987
}
```

Type conversions IV

- C++ ist stark typisiert
- Einige Konvertierungen erfordern eine explizite Angabe.

```
double x = 10.3;
int y;
y = int (x);     // functional notation
y = (int) x;     // c-like cast notation
```

Type conversions V

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Die Umwandlung von Strings in Zahlen ist kein Typecast...

```
// string
char string[] = "938";

// char -> int
int i = atoi( string );

// char -> double
double d = atof( string );
```

Type conversions VI

- Die Umwandlung von Zahlen in Strings ist. . . kompliziert
- Viele Lösungen. Hier: Stringstream. Eine String wird wie `cout` verwendet

```
// stringstream::str
#include <string>       // std::string
#include <iostream>     // std::cout
#include <sstream>      // std::stringstream, std::stringbuf

int main () {
        int i = 17;
        std::string s;
  std::stringstream ss;
  ss << i;
        ss >> s;
  std::cout << s << std::endl;
  return 0;
}
```

Fragen / Unklarheiten in C++ bis jetzt?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?
- Dabei Dateien verloren, weil zwei Leute gleichzeitig gespeichert haben?

Was ist ein Version Control System

Ein Version Control System

- Speichert Schnapschüsse (Commits) eines Verzeichnisses (mit Unterverzeichnissen)
- Speichert für jeden Commit eine Beschreibung, was sich geändert hat und wer es geändert hat
- Ermöglicht es, für Textdateien genau anzuzeigen, wie sich zwei Versionen einer Datei unterscheiden
- Speichert (oft) eine Kopie der Daten auf einem Server
  - Datensicherung
  - Datenaustausch mit anderen Computern, Entwicklern
- Erstellt Commit nur auf explizite Anfrage
  - Keine kaputten Versionen committen (kompiliert nicht etc.)
  - Commit-Beschreibung muss eingegeben werden
- Unterstützt dabei, gleichzeitige Änderungen von mehreren Entwicklern zusammenzuführen (merge)

git — Distributed Version Control System

- Kommandozeilenprogramm zum Verwalten von (ursprünglich) text-basierten Dateien
- Entwickelt 2005 von Linus Torvalds für die Quellen des Linux-Kernels
- Extrem schnell
- Verwaltet einige der grössten Codebasen weltweit:
  - Linux-Kernel ($>$ 25 Mio. Zeilen, $>$ 10.000 Commits / Version)
  - Windows (300 GB, 3,5 Mio. Dateien)
- Kostenloses Repository-Hosting, z.B. GitHub, Bitbucket, GitLab
- Unterstützung für Bilder etc. mit git-lfs
- Inzwischen de-facto Industriestandard

Use Cases

Warum machen wir das ganze hier?

- Wichtig, um später an realen Softwareprojekten mitarbeiten zu können
- Braucht eine gewisse Eingewöhnungsphase
- Konsequente Verwendung von Versionskontrolle hilft beim Strukturieren der Programmierarbeit
- Einfache Möglichkeit, zusammen an Übungen zu arbeiten und diese abzugeben.

Glossar

Repository Datenbank mit allen Informationen über Dateiversionen in einem Projekt, liegt im versteckten Verzeichnis `.git` im obersten Projektverzeichnis.

Commit Globaler Schnappschuss aller Projektdateien mit einer Beschreibung der Änderungen zur vorherigen Version.

Branch Eine Abfolge von Commits, die einen Entwicklungszweig abbilden. Ein Repository kann mehrere Branches enthalten. Der Standard-Branch heißt `master`.

Tag Ein dauerhafter Name für einen Commit, z.B. für ein Release.

Globale Konfiguration

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Zuerst sollten wir zwei Dinge einrichten:

- git möchte wissen, wer wir sind:

```
git config --global user.name "Felix Spanier"
git config --global user.email "felix.spanier@uni-heidelberg.de"
```

- Für Git-Status in der Kommandozeile folgende Zeilen in `~/.bash_profile` einfügen:

```
export PS1='[\u@\h \W$(__git_ps1 " (%s)")]\$ '
export GIT_PS1_SHOWDIRTYSTATE=1
```

Getting Started

Zum Anlegen des Repositories ins oberste Projektverzeichnis wechseln und dann:

```
[git-tutorial]$ git init
Initialized empty Git repository in /home/fspanier/git-tutorial/.git/
[git-tutorial (master #)]$
```

Damit existiert das Repository, aber es gibt noch keinen Commit:

```
[git-tutorial (master #)]$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.cc
        input.hh

nothing added to commit but untracked files present (use "git add" to track)
[git-tutorial (master #)]$
```

- Kopieren Sie beliebige Dateien der ersten Wochen in den Ordner, wenn Sie die Befehle ausprobieren möchten

Änderungen hinzufügen

git speichert nur Änderungen, von denen wir ihm explizit erzählen:

```
[git-tutorial (master #)]$ git add input.cc
[git-tutorial (master +)]$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   input.cc

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.hh

[git-tutorial (master +)]$
```

Änderungen committen

Jetzt können wir auch noch input.hh hinzufügen und einen Commit erzeugen:

```
[git-tutorial (master +)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master (root-commit) 1bb9ef8] Added input files
 2 files changed, 25 insertions(+)
 create mode 100644 input.cc
 create mode 100644 input.hh
[git-tutorial (master)]$ git status
On branch master
nothing to commit, working tree clean
[fspanier@ipk git-tutorial (master)]$ git log
commit 1bb9ef87e4c235cc72e07009fc48cefb38df6154 (HEAD -> master)
Author: Felix Spanier <felix.spanier@uni-heidelberg.de>
Date:   Fri Dec 1 10:55:17 2017 +0100

    Added input files
  [fspanier@ipk git-tutorial (master)]$
```

Commits

- Commits enthalten
  - einen Snapshot aller Dateien,
  - den Erstellungszeitpunkt,
  - Namen und Inhalt vom Autor der Änderungen und von der Person, die den Commit erstellt hat,
  - eine Beschreibung der Änderungen (Changelog),
  - Eine Liste mit Verweisen auf die Eltern-Commits.
- Commits werden durch einen Hash (eine Prüfsumme) ihres Inhalts identifiziert, z.B. `1bb9ef87e4c235cc72e07009fc48cefb38df6154`.
- Commit-Hashes können abgekürzt werden, solange sie eindeutig sind.
- Commits können nicht verändert werden:
  anderer Inhalt $\Rightarrow$ anderer Hash.

Weiterarbeiten

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Wir verändern die Datei `input.hh` und speichern die veränderte Datei:

```
[git-tutorial (master *)]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   input.hh

no changes added to commit (use "git add" and/or "git commit -a")
[git-tutorial (master *)]$
```

Zur Erinnerung: Änderungen müssen wir git immer mitteilen!

```
[git-tutorial (master *)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master bab868d] Added comment
 1 file changed, 2 insertions(+)
[git-tutorial (master)]$
```

Unterschiede anzeigen

Anzeigen, was der aktuelle Commit verändert hat:

```
[git-tutorial (master)]$ git show master
commit bab868dd7e345f1b660157a8bd4519cad175733d (HEAD -> master)
Author: Felix Spanier <felix.spanier@uni-heidelberg.de>
Date:   Fri Dec 1 11:06:27 2017 +0100

    Added comment

diff --git a/input.hh b/input.hh
index 3b74a4a..3bef25c 100644
--- a/input.hh
+++ b/input.hh
@@ -4,6 +4,8 @@
 #include <string>
 #include <istream>

+// Reads from input until EOF and returns the
+// result as a string.
 std::string read_stream(std::istream& input);

 #endif // INPUT_HH
[git-tutorial (master)]$
```

Tipp

Diffs mit grafischen Hilfsprogrammen (z.B. meld) oder einfach auf dem Server anschauen!

Branches

- Ein Branch ist ein Entwicklungszweig innerhalb eines Repositories.
- Repositories können beliebig viele Branches enthalten.
- `git status` sagt einem, auf welchem Branch man sich befindet.
- Ein Branch zeigt auf einen Commit.
- Wenn man einen neuen Commit erstellt, speichert er den aktuellen Commit als Vater und der Branch zeigt danach auf den neuen Commit.

Branches: Befehle

- Branch `playground` erstellen:

```
git branch playground
```

- Branches auflisten:

```
[git-tutorial (master)]$ git branch
* master
  playground
[git-tutorial (master)]$
```

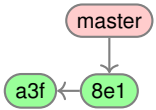
- Branch wechseln:

```
[git-tutorial (master)]$ git checkout playground
Switched to branch 'playground'
[git-tutorial (playground)]$
```

- Änderungen von anderem Branch importieren (merge):

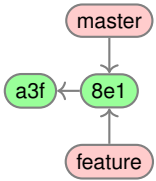
```
git merge other-branch
```

Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

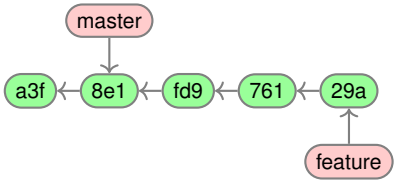
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen

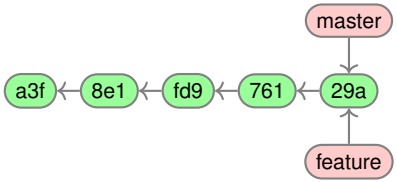
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen
- Auf Branch arbeiten

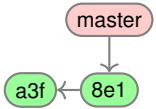
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

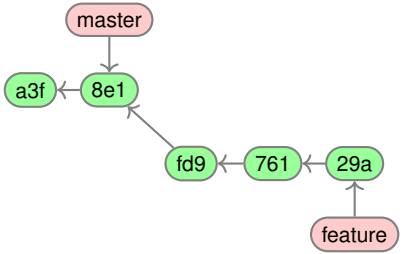
- Branch anlegen
- Auf Branch arbeiten
- Keine neuen Commits in master $\Rightarrow$ fast-forward

Merging II



Der realistische Fall: Gleichzeitige Änderungen

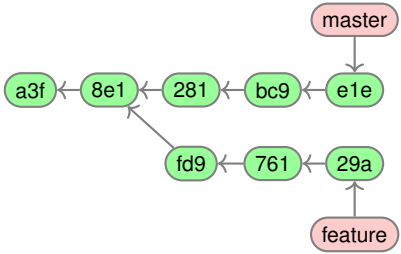
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten

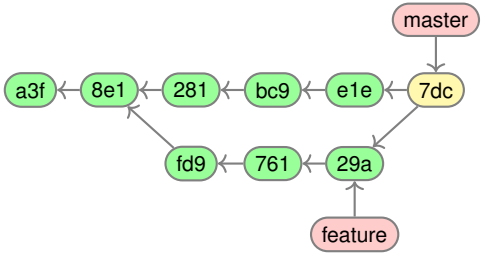
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert

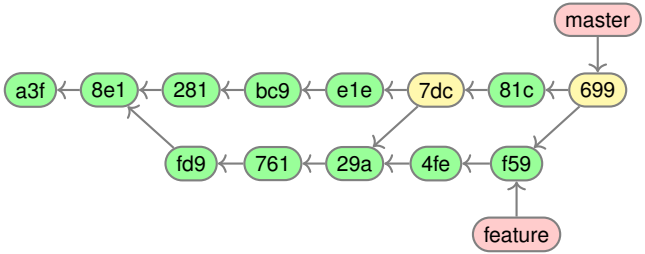
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Mehrere Repositories I

- git kann Änderungen zwischen Repositories synchronisieren.
- Zusätzliche Repositories heißen remote.
- Branches aus einem Remote-Repository bekommen den Namen des Repositories vorangestellt.
- Man kann ein Remote-Repository klonen, um den Inhalt zu bekommen:

```
[folder]$ git clone https://gitlab.dune-project.org/core/dune-common.git
Cloning into 'dune-common'...
remote: Counting objects: 54485, done.
remote: Compressing objects: 100% (13788/13788), done.
remote: Total 54485 (delta 40840), reused 54059 (delta 40531)
Receiving objects: 100% (54485/54485), 12.83 MiB | 19.15 MiB/s, done.
Resolving deltas: 100% (40840/40840), done.
[folder]$
```

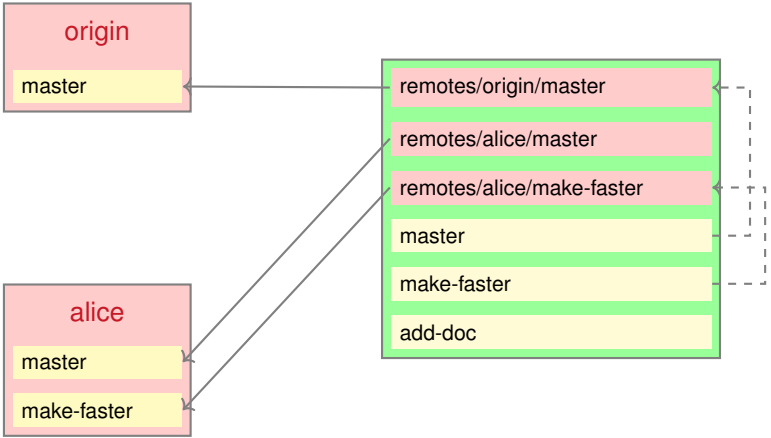
Mehrere Repositories II

- Man kann Branches von Remote-Repositories nicht direkt auschecken.
- Git legt beim ersten checkout einen tracking branch an:

```
[dune-common (master)]$ git checkout releases/2.6
Branch 'releases/2.6' set up to track remote branch 'releases/2.6' from 'origin'.
Switched to a new branch 'releases/2.6'
[dune-common (releases/2.6)]$
```

- Neue Änderungen können mit `git pull` heruntergeladen und gemergt werden.
- Eigene Änderungen können mit `git push` hochgeladen werden.
  - Wenn jemand anderes vorher Änderungen hochgeladen hat: Fehler!
  - Lösung: Änderungen erst mit `git pull` herunterladen und integrieren, dann nochmal pushen.

Mehrere Repositories — Schema



Git — Wichtige Befehle

init	Leeres Repository anlegen
clone	Bestehendes Repository klonen
add	Änderungen für Commit registrieren
commit	Commit erstellen
log	Verlauf ansehen
diff	Änderungen ansehen
branch	Branches anlegen, auflisten, löschen
checkout	Branch wechseln
merge	Branches zusammenführen
pull	Neue Änderungen herunterladen
push	Neue Änderungen hochladen

```
git help <Befehl> für Hilfe!
```

Credential Caching in git

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Es gibt zwei Möglichkeiten, das ständige Eingeben von Passwörtern in git zu umgehen:

- Erstellen eines SSH-Schlüssels, Hochladen des Schlüssels auf den Server (über das Webinterface) und aktivieren des SSH agents (siehe Tutorials im Internet, SSH Agent hängt von der Linux-Distribution ab)
- Aktivieren des Credential Cache:

```
git config --global credential.helper cache
```

Durch dieses Kommando wird git auf dem lokalen Rechner so konfiguriert, dass Benutzername und Passwort für 15 min gespeichert werden

C++-Standardbibliothek

- C++ enthält eine umfangreiche Standardbibliothek mit vielen Datenstrukturen und Algorithmen.
- Wenn möglich, ist es **immer** besser, Funktionen aus der Standardbibliothek zu nehmen als eigene.
  - Gut optimiert
  - Umfangreich getestet
- Bestandteile:
  - Datenstrukturen
  - Algorithmen
  - Mathematische Funktionen
  - Input / Output
- Gute Referenz auf `https://cppreference.com`

Coding Style

- Bei der Verwendung der Standardbibliothek sollte das `std::` normalerweise explizit hingeschrieben werden.
- In besonderen Fällen ist es manchmal erforderlich, eine Funktion ohne `std::` aufzurufen. Diese sollte dann lokal importiert werden:

```
void foo()
{
  int a, b;
  using std::swap;
  swap(a,b);
}
```

- Die Verwendung `using namespace std;` ist mit vorsicht zu genießen und wird in größeren Projekten als schlechte Praxis betrachtet.
  - Schwierig zu sagen, woher eine Funktion kommt
  - Kann zu sehr subtilen Fehlern führen, die schwer zu debuggen sind

Container (Datenstrukturen)

Container bzw. Datenstrukturen verwalten eine zusammenhängende Menge von Werten mit bestimmten Eigenschaften. C++ liefert einige praktische Container mit:

`array`	für Listen von Werten, deren Anzahl zur Compile-Zeit bekannt ist
`vector`	für Listen von Werten, deren Anzahl zur Compile-Zeit nicht bekannt ist
`list`	für Listen von Werten, bei denen oft in der Mitte Elemente hinzufügt oder entfernt werden
`(unordered_)map`	für das Zuordnen von Werten zu Schlüsseln beliebigen Typs (z.B. für ein Wörterbuch). Es gibt eine Variante, die die Eintrage nach den Schlüsseln sortiert, und eine, die das nicht tut.
`pair`	Ein Paar von Werten, die unterschiedliche Typen haben können
`tuple`	Eine Liste von Werten, die unterschiedliche Typen haben können

array

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- Liste, deren Länge zur Compile-Zeit bekannt ist:

```
std::array<Datentyp,Länge>
```

- Einfachster Typ, der das C++-Container-Interface erfüllt
- Beispiel:

```
#include <array>
#include <iostream>

int main(int argc, char** argv)
{
  std::array<int,4> a = {{1,2,3,4}};
  std::cout << a.size() << std::endl; // 4
  a[2] = 4;
  std::cout << a[3];
}
```

- Manchmal sieht man auch C arrays (`int a[4];`), aber besser C++-Version verwenden.

Eigenschaften von Containern

- Container sind Objekte (mehr dazu später).
  - Objekte haben member variables und member functions, die zu dem jeweiligen Objekt gehören und darin gespeichert werden oder auf dieses wirken:

```
p.first = 3; // member variable
a.size();    // member function
```

  - Member function werden auch Methoden genannt.
- Container sind Templates. Templates sind parametrisierte Typen, die als Templateparameter in spitzen Klammern andere Typen oder Konstanten übergeben bekommen.
  - Bei Containern wird hierüber z.B. festgelegt, welchen Typ von Werten sie speichern können.
  - Mehr zu Templates später.
- Der Zugriff auf Inhalte eines Containers erfolgt meistens mit eckigen Klammern:

```
my_array[3];
my_map["key"];
```

vector

Dynamisch anpassbare Liste von Objekten:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
  std::vector<int> a; // leere Liste
  std::vector<int> b = {{1,2,3,4}};
  std::vector<int> c(20); // Liste mit 20 Einträgen
  std::cout << b.size() << std::endl; // 4
  b[2] = 4;
  a = b; // kopiert den Inhalt
  a.resize(100); // Grösse anpassen
  a.push_back(1); // Wert 1 hinten anfügen
  a.pop_back();   // Letztes Element entfernen
}
```

- Bei grossen Listen ($>$ 100–1000) immer besser als `std::array`.

Iterieren über Container

```
std::vector<int> v = {{1,2,3,4}};
for (int i = 0 ; i < v.size() ; ++i)
  std::cout << v[i] << std::endl;
```

- Aufpassen am Ende! Index wird nicht auf Gültigkeit überprüft.
- Prüfen kostet unnötig viel Zeit, wenn für jeden Eintrag nur wenige Operationen benötigt werden, z.B. Summe
- Bei Fehlern: Ersetze `v[i]` durch `v.at(i)`, gleiche Bedeutung, aber prüft Gültigkeit

Iterieren über Container


```
std::vector<int> v = {{1,2,3,4}};
for (int i = 0 ; i < v.size() ; ++i)
  std::cout << v[i] << std::endl;
```

- Aufpassen am Ende! Index wird nicht auf Gültigkeit überprüft.
- Prüfen kostet unnötig viel Zeit, wenn für jeden Eintrag nur wenige Operationen benötigt werden, z.B. Summe
- Bei Fehlern: Ersetze `v[i]` durch `v.at(i)`, gleiche Bedeutung, aber prüft Gültigkeit

Vereinfachter Loop zum Iterieren über Standard-Container:

```
std::vector<int> v = {{1,2,3,4}};
for (int entry : v)
  std::cout << entry << std::endl;
```

- Funktionert für alle Standard-Container.
- Besser lesbar.
- Keine Gefahr, Fehler am Ende zu machen ($<$ vs. $\leq$).

pair und tuple

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- `pair` speichert genau zwei Werte. Oft benutzt, um zwei Werte aus einer Funktion zurückzugeben:

```
#include <utility>

std::pair<std::string,int> nameAndGrade(int matrikelNr) {
  return std::make_pair("Max",2);
}

std::pair<std::string,int> r = nameAndGrade(42);
std::cout << "Name: " << r.first << std::endl
          << "Note: " << r.second << std::endl;
```

pair und tuple

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- `tuple` speichert mehrere Werte. Etwas komplizierterer Zugriff:

```
#include <tuple>

std::tuple<int,int,int> birthDay(int matrikelNr) {
  return std::make_tuple(1,1,1990);
}

std::tuple<int,int,int> bday = birthday(42);
std::cout << "Tag: " << std::get<0>(bday) << std::endl
          << "Monat: " << std::get<1>(bday) << std::endl
          << "Jahr: " << std::get<2>(bday) << std::endl;
```

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Listen fixer Länge:
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit “Löchern” in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Listen fixer Länge:
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit “Löchern” in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind
- Lösung: Maps als Abbildung von Werten mit Typ `Key` auf Werte mit Typ `Value`:
  - `std::map<Key,Value>` speichert Einträge in sortierter `Key`-Reihenfolge.
  - `std::unordered_map<Key,Value>` speichert Einträge in zufälliger Reihenfolge, ist bei vielen Schlüsseln deutlich schneller.

Maps: Syntax I

- Benötigen `#include<map>` bzw. `<unordered_map>`
- Verwendung identisch, im folgenden nur für `std::map` gezeigt.
- Maps werden immer leer angelegt:

```
std::map <std::string,int> shopping_list;
```

- Einträge werden beim ersten Zugriff angelegt:

```
shopping_list["cookies"] = 3; // create or overwrite value
shopping_list.insert({"cookies",3}); // unsuccessful if key "cookies" exists
```

- Fehlende Einträge werden beim Abfragen mit dem Standardwert (bei Zahlen: 0) initialisiert:

```
shopping_list["biscuits"]; // returns 0
```

- Die Grösse der Map kann wieder mit `size()` bestimmt werden:

```
shopping_list.size(); // returns 2 (cookies and biscuits)
```

Maps: Syntax II

- Testen, ob ein Eintrag in der Map enthalten ist:

```
// returns 1, as there is 1 entry for key biscuits
shopping_list.count("biscuits");
// returns 0, as there is no entry for key crisps
shopping_list.count("crisps");
```

  Hier wird eine Anzahl zurückgegeben, weil die Bibliothek auch Multi-Maps enthält, die einem Schlüssel mehrere Einträge zuordnen können.

- Eintrag löschen:

```
// returns 1, because 1 element removed
shopping_list.erase("biscuits");
// returns 0, because 0 elements removed
shopping_list.erase("crisps");
```

- Map komplett leeren:

```
shopping_list.clear();
```

Sortieren

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- C++ hat einen hochoptimierten, eingebauten Sortier-Algorithmus.
- Der Algorithmus basiert auf *Iteratoren*, was im Moment aber bis auf die Syntax zum Aufrufen egal ist:

```
#include <vector>
#include <algorithm>

int main(int argc, char** argv)
{
  std::vector<int> a = .....;
  // sortiert a nach aufsteigenden Zahlenwerten
  std::sort(a.begin(),a.end());
}
```

Variablen (Wiederholung)

- Variablen repräsentieren eine Stelle im Arbeitsspeicher, an der Daten eines bestimmten Typs gespeichert sind.
- Jede Variable hat einen Namen und einen Typ.
- Der Speicherbedarf einer Variablen
  - hängt von ihrem Typ ab.
  - kann mit dem Operator `sizeof(var)` oder `sizeof(type)` abgefragt werden.
- Die Stelle im Speicher, an der der Wert einer Variablen gespeichert ist, kann nicht verändert werden.

Konstante Variablen

- Variablen in C++ können mit dem Keyword **`const`** als konstant (unveränderlich) deklariert werden.
- Eine konstante Variable kann nicht mehr verändert werden, nachdem sie definiert wurde:

```
const double pi = 3.1415926535;
pi = pi + 1; // compile error
```

- Konstante Variablen können helfen, Programmierfehler zu vermeiden.
- Unter bestimmten Umständen kann der Compiler schnelleren Code generieren, wenn Variablen konstant sind.

Referenzen

- Referenzen sind zusätzliche Namen für existierende Variablen.
- Der Typ einer Referenz ist der Typ der existierenden Variablen gefolgt von `&`. Der Typ einer Referenz auf `int` ist also `int&`.
- Eine Referenz wird **immer** in dem Moment initialisiert, in dem sie definiert wird:

```
int x = 4;
int& x_ref = x;
int& no_ref; // compile error!
```

- Eine Referenz zeigt immer auf die gleiche Variable.
- Die Referenz verhält sich genau so wie die Original-Variable.
- Änderungen an der Referenz verändern auch die Original-Variable und umgekehrt.

Referenzen: Beispiel

```cpp
# include <iostream>

int main ()
{
  int a = 12;
  int& b = a; // definiert Referenz
  int& c = b; // Referenz auf Referenz
  float& d = a; // falscher Typ, Compile-Fehler
  int e = b;
  b = 2;
  c = a * b;
  std::cout << a << std::endl; // 4
  std::cout << e << std::endl; // 12
}
```

Iterieren über Container mit Referenzen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Wenn man den Inhalt eines Containers beim Iterieren verändern will, muss man für die Variable einen Referenz-Typ verwenden:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
        std::vector<int> v = {1,2,3,4};
        for (int& value : v)
        value *= 2;
}
```

Iterieren über Maps (mit Referenzen)

- Beim Iterieren möchten wir auf Schlüssel und zugeordneten Wert zugreifen können.
- Verbesserter `for`-Loop liefert Referenz auf `std::pair<const Key,Value>` zurück:

```
for (std::pair<const std::string,int>& entry : shopping_list)
std::cout << entry.first << ": "
<< entry.second << std::endl;
```

- Bei Verwendung von `std::map` werden die Einträge nach aufsteigender `Key`-Reihenfolge sortiert abgelaufen.
- Bei Verwendung von `std::unordered_map` ist die Reihenfolge der Einträge zufällig.
- Für Fortgeschrittene: `std::map` basiert auf einem sortierten Binärbaum, `std::unordered_map` auf einer Hashtable.

Call by Value

- Wenn eine Funktion mit einem normalen Parameter aufgerufen wird, erstellt C++ eine Kopie des Parameterwerts, mit dem die Funktion dann arbeitet:

```
double square(double x);
```

- Diese Aufrufkonvention heißt *call by value*.
- Bisher haben wir in den Übungen nur call by value verwendet.
- Weil die Funktion eine Kopie der Original-Variablen bekommen hat, wirken sich Änderungen in der Funktion nicht auf die Original-Variable aus.

Call by Reference

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Wenn eine Funktion mit einem Referenz-Parameter aufgerufen wird, übergibt C++ eine Referenz auf die Original-Variable an die Funktion:

```
void sort(std::vector<int>& x);
```

- Diese Aufrufkonvention heißt *call by reference*.
- Funktionen mit dieser Aufrufkonvention können Werte außerhalb der Funktion verändern, ohne dass dies beim Aufruf direkt ersichtlich ist (sie haben *Seiteneffekte*).
- Nicht-triviale Datentypen wie `std::vector` sollten normalerweise per Referenz übergeben werden, weil das Kopieren teuer sein kann.

Dangling References

- Intern sind Referenzen eine spezielle Art von konstanter Variable, die den Speicherort der Original-Variablen enthält.
- Wenn die Original-Variable aufhört zu existieren, wird ihr Speicherort ungültig.
- Referenzen auf die nicht mehr existierende Variable greifen weiter auf den ungültigen Speicherort zu
  $\Rightarrow$ Programm stürzt ab oder liefert falsches Ergebnis!
- Tipps für Referenzen:
  - Referenzen sind oft gut für Funktionsparameter (call by reference).
  - Niemals Referenzen als Rückgabewert von normalen Funktionen verwenden!

Keyword `auto`

- Typnamen oft lang und umständlich:

```
for (const std::pair<std::string,int>& entry : shopping_list)
  std::cout << entry.first << ": "
            << entry.second << std::endl;
```

- Typ der Einträge ist Compiler bekannt
- `auto` *rät* Variablentypen basierend auf der rechten Seite der Zuweisung:

```
auto cookies = shopping_list["cookies"]; // auto -> int
```

  - Standardmäßig immer value type (kopiert Rückgabewert).
  - Nach Bedarf mit `&` und `const` qualifizieren.
- Beispiel:

```
for (auto& entry : shopping_list)
  std::cout << entry.first << ": "
  << entry.second << std::endl;
```

Control Flow in Schleifen

- Manchmal ist es nötig, eine Schleife vorzeitig zu verlassen. Hierfür gibt es das Keyword **`break`**:

```
for (int step = 0 ; step < steps ; ++step) {
  bool ok = do_step(step);
  if (not ok)
    break;  // leaves the loop, skipping later steps
}
```

- Manchmal ist es nötig, eine Iteration der Schleife vorzeitig zu beenden und direkt zur nächsten zu springen. Hierfür gibt es das Keyword **`continue`**:

```
for (auto& sweets : shopping_list) {
  if (sweets.second == 0) {
    // we don't really have those sweets in the list
    continue; // jump to next list entry
  }
  put_in_basket(sweets);
}
```

Exceptions

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Wenn C++ Code ausgeführt wird, können aus den verschiedensten Gründen Fehler auftreten.
- In C++ wird in der Regel versucht eine sinnvolle Fehlermeldung zu produzieren. Dafür werden oft *Exceptions* genutzt.

```
int main() {
  vector<int> v(10);
  // error: terminating with uncaught exception
  // of type std::out_of_range: vector
  cout << v.at(10);
}
```

- Um nicht bei jedem Fehler einen Programmabsturz zu bekommen gibt es die Möglichkeit Fehler abzufangen.

Try / Catch

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Mit einem `try {} catch {}` Block können Exceptions vom Programmierer abgefangen werden:

```
int main() {
  vector<int> v(10);
  try {
    cout << v.at(10);
  } catch (...) { // Catches All types of exceptions
          cout << "You made a mistake.. dummy!\n";
  }
}
```

Throw

- Exceptions können auch selbst mit **throw** erzeugt werden.
- Catch kann auch nur bestimmte Exception-Typen fangen

```
void print_item(int i, vector<int> v) {
if (i <= v.size())
  throw OutOfRangeException("i is out of range!");
}
...
try {
  print_item(1000, {1, 2, 3, 4})
} catch (OutOfRangeException e) { // catch only OutOfRangeException
...
}
```

C++-Projekte jenseits kleiner Übungen

Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, . . .).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

C++-Projekte jenseits kleiner Übungen

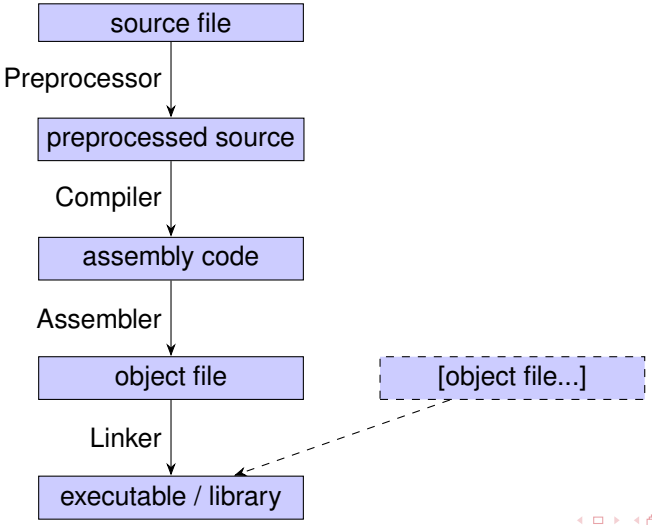
Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, . . .).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

- Verständnis des Kompilier-/Buildprozesses
- Aufteilung von Code auf mehrere Dateien
- Verwaltung der Abhängigkeiten zwischen Dateien

Der C++-Kompilierprozess



Der Präprozessor

- Der C++-Präprozessor fügt Header-Dateien in Quellcode ein und expandiert Makros.
- Alle Zeilen, die mit `#` anfangen (sogenannte Direktiven), werden vom Präprozessor verarbeitet.
- Die wichtigsten Direktiven:
  - `#include <header>` fügt den Inhalt der Datei `header` an dieser Stelle ein.
  - `#include "header"` fügt den Inhalt der Datei `header` an dieser Stelle ein, sucht die Datei aber auch im aktuellen Verzeichnis.
  - `#define MACRO REPLACEMENT` definiert ein Makro: Immer, wenn nach dieser Zeile `MACRO` als alleinstehendes Wort auftaucht, wird es durch `REPLACEMENT` ersetzt.
  - Text zwischen `#ifdef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro nicht definiert ist.
  - Text zwischen `#ifndef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro definiert ist.
- Der Präprozessor kann mit `g++ -E` ausgeführt werden.

Der Compiler

- Der Compiler übersetzt den C++-Code in einfachere Befehle, die der Prozessor verstehen kann.
- Das Resultat dieses Schritts ist Assembly-Code, eine für Menschen lesbare Version der Maschinenbefehle.
- Assembly Code enthält keinerlei Variablennamen oder Schleifen mehr.
- Die Ausgabe des Compilers unterscheidet sich je nach Prozessor (Smartphone-Prozessoren verwenden andere Befehle als PCs).
- Der Compiler kann in diesem Schritt das Programm stark optimieren, wenn aktiviert (Option `-O2` oder `-O3`).
- Die Ausgabe des Compilers kann man mit `g++ -S` oder auf `https://godbolt.org` anschauen.

Der Assembler

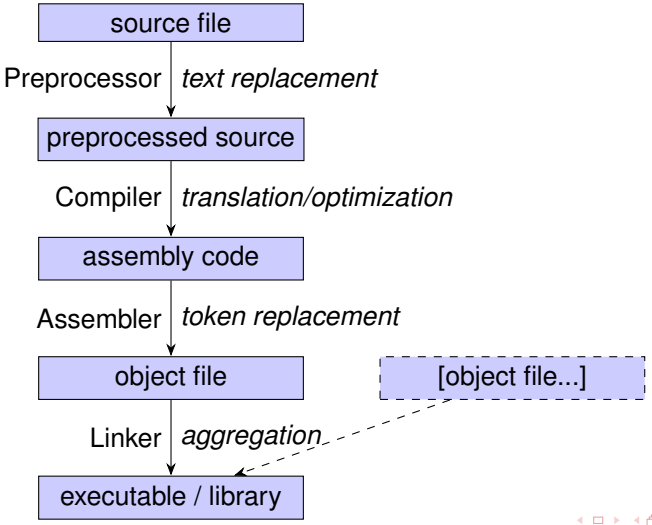
- Der Assembler verwandelt Assembly-Code in die binären Befehlscodes, die der Prozessor versteht.
- Der Assembler produziert sogenannte *object files* mit der Erweiterung `.o`.
- Achtung: das hat nichts mit den “Objekten” aus der objektorientierten Programmierung zu tun!
- Object files enthalten *object code*, das Endprodukt des Kompiliervorgangs im engeren Sinne.
- Um ein object file zu erzeugen, muss der Compiler mit der Option `-c` aufgerufen werden.

Der Linker

- Der Linker kombiniert den object code aus einem oder mehreren object files und Programmbibliotheken und erzeugt das Endprodukt des Build-Prozesses:
  - Ausführbare Dateien (executables), die von der Kommandozeile aufgerufen werden können.
  - Bibliotheken (libraries), die Funktionen enthalten und diese für andere Programme / Bibliotheken zur Verfügung stellen.
- Funktionen aus einigen Standardbibliotheken werden vom Linker automatisch gefunden, andere muss man explizit angeben.
- Linkeraufruf, um ein ausführbares Programm aus mehreren object files zu erzeugen:

```
g++ -o executable file1.o file2.o ...
```

Der C++-Kompilierprozess Revisited



Programme mit mehreren Dateien

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
double cube(double x)
{
  return x * x * x;
}
```

- Funktionen, die man mehrfach verwendet, sollte man in eine eigene Datei auslagern:
  - Einfache Wiederverwendbarkeit.
  - Bessere Übersichtlichkeit bei grösseren Programmen.
- Man benötigt meistens zwei Dateien:
  - Immer ein *header file*, das von anderen Dateien eingebunden werden kann und alle Funktionalität, die wir bereitstellen, *deklariert*.
  - Ein *implementation file*, das die eigentliche Implementierung enthält. Dies kann in manchen Situationen (siehe Templates) entfallen.

Kompilieren des Projekts


```
g++ -Wall -std=c++14 -c cube.cc
g++ -Wall -std=c++14 -c main.cc
g++ -Wall -o example cube.o main.o
```

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

Kompilieren des Projekts

```
g++ -Wall -std=c++14 -c cube.cc
g++ -Wall -std=c++14 -c main.cc
g++ -Wall -o example cube.o main.o
```

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

$\Rightarrow$ Automatisierung des Prozesses durch Buildsysteme (make, CMake, qmake, autotools, . . .)

CMake

- CMake ist ein leistungsfähiges Buildsystem für Projekte in C und C++.
- Abhängigkeiten zwischen Programmen, Quell- und Headerdateien werden automatisch erkannt.
- CMake kann testen, ob das Betriebssystem bestimmte Features hat, und den Buildprozess daran anpassen.
- CMake unterstützt unterschiedliche Build-Konfigurationen:
  - Debug (für Entwicklung und Fehlersuche)
  - Release (generiert schnellere Programme für die spätere Nutzung: aktiviert Optimierung)
- CMake trennt sauber zwischen
  - Quellcode-Verzeichnis (enthält `.cc`-Dateien etc.)
  - Build-Verzeichnis (enthält alles, was automatisch generiert wird, z.B. Programme)
- CMake ist streng genommen ein Build System Generator, es erzeugt eine Konfiguration für andere Build Systems, die dann für das eigentliche Bauen verwendet werden.

CMakeLists.txt

- CMake wird über Dateien mit dem festen Namen `CMakeLists.txt` konfiguriert.
- Dateien beschreiben, wie das Projekt konfiguriert werden soll und welche Programme und Bibliotheken aus welchen `.cc`-Dateien gebaut werden sollen.
- Minimalbeispiel:

```
# Set minimum required CMake version
cmake_minimum_required(VERSION 3.5)
# Start project and set its name to ipk-demo
project(ipk-demo LANGUAGES CXX)

# Force compiler to run in C++14 mode
set(CMAKE_CXX_STANDARD 14)

# Create executable programs
add_executable(cube cubemain.cc cube.cc)
add_executable(calculator calcmain.cc basic.cc cube.cc)
```

CMake verwenden

- Im ersten Schritt erzeugt man mit CMake ein Buildsystem für `make`, indem man `cmake <pfad-zum-verzeichnis-mit-cmakelists.txt>` aufruft.
- Das Buildsystem muss in einem anderem Verzeichnis erzeugt werden als die Quelldateien.
- Eine gute Wahl ist das Unterverzeichnis `build/`:

```
mkdir build
cd build
cmake ..
```

- Wenn das Buildsystem existiert, startet man den eigentlichen Build-Prozess mit dem Befehl `make` im Verzeichnis, in dem man auch `cmake` aufgerufen hat (hier: `build/`).

CMake-Feintuning

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Um genauer zu steuern, wie CMake ein Projekt baut, kann man dem CMake-Aufruf Variablen mitgeben:

```
cmake -DVARIABLE=VALUE <pfad>
```

Wichtige Variablen sind:

`CMAKE_CXX_COMPILER`	Der gewünschte C++-Compiler (g++, clang++, etc.)
`CMAKE_CXX_FLAGS`	Zusätzliche Flags für den Compiler, z.B. `-Wall` etc.
`CMAKE_BUILD_TYPE`	Build-Konfiguration (Release oder Debug)

Weitere Optionen kann man finden, indem man nach `cmake` im Build-Verzeichnis `ccmake .` aufruft, die Taste `t` drückt und dann durch die Liste blättert.

Wichtige CMake-Befehle

- Eine ausführbare Datei anlegen:

```
add_executable(<name> <.cc-Datei>...)
```

- Eine Bibliothek anlegen:

```
add_library(<name> <.cc-Datei>...)
```

- Target (executable oder library) gegen eine Bibliothek linken:

```
target_link_libraries(<target> PUBLIC <library>...)
```

- Unterstützung für automatische Tests aktivieren:

```
enable_testing()
```

- Test anlegen:

```
add_executable(calculator_test calculator_test.cc...)
```

Tests mit CMake

- Tests sind ein essentieller Bestandteil guter Programmierung!
- Tests prüfen, ob eine Funktion für bestimmte Inputs das erwartete Resultat produziert.
- In CMake ist ein Test ein normales Programm, das sich an folgende Konvention hält:
  - Wenn der Test erfolgreich war, gibt `main()` 0 zurück.
  - Wenn der Test fehlgeschlagen ist, gibt `main()` $0 < n < 127$ zurück.
- Vor dem Anlegen von Tests muss man `enable_testing()` aufrufen.
- Mit dem Befehl `ctest` führt CMake alle Tests aus und zeigt die Resultate auf der Konsole an.

git best Practices

- Keine Erzeugten Dateien in git hinzufügen

git best Practices

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Keine Erzeugten Dateien in git hinzufügen
  $\rightarrow$ Oft nur Temporär / System u. Plattform spezifisch

git best Practices

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Keine Erzeugten Dateien in git hinzufügen
  $\rightarrow$ Oft nur Temporär / System u. Plattform spezifisch
- Häufig und kleinschrittig commits erstellen

git best Practices

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Keine Erzeugten Dateien in git hinzufügen
  $\rightarrow$ Oft nur Temporär / System u. Plattform spezifisch
- Häufig und kleinschrittig commits erstellen
- Sinnvolle commit Nachrichten schreiben

git best Practices

- Keine Erzeugten Dateien in git hinzufügen
  $\rightarrow$ Oft nur Temporär / System u. Plattform spezifisch
- Häufig und kleinschrittig commits erstellen
- Sinnvolle commit Nachrichten schreiben
- Vor dem commit auf erfolgreiche Kompilierung prüfen

git best Practices

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Keine Erzeugten Dateien in git hinzufügen
  $\rightarrow$ Oft nur Temporär / System u. Plattform spezifisch
- Häufig und kleinschrittig commits erstellen
- Sinnvolle commit Nachrichten schreiben
- Vor dem commit auf erfolgreiche Kompilierung prüfen
- Nutze `.gitignore`
  - https://github.com/github/gitignore
  - Vorlagen für viele verschiedene Projekte

Objektorientierte Programmierung

Bisher:

- Programme aus Funktionen, die mit primitiven Datentypen arbeiten (Zahlen, Strings)
- Verwendung von komplexeren Datentypen aus der STL

Objektorientierte Programmierung

Bisher:

- Programme aus Funktionen, die mit primitiven Datentypen arbeiten (Zahlen, Strings)
- Verwendung von komplexeren Datentypen aus der STL

Jetzt:

- Programme aus Objekten, die
  - einen internen Zustand haben (member variables)
  - Operationen ausführen können (member functions / Methoden)
- Jedes Objekt ist eine *Instanz* einer *Klasse*.
- Eine Klasse definiert das Verhalten all ihrer Instanzen.
- Wichtige Konzepte:
  - Kapselung
  - `const`ness
  - Komposition vs. Vererbung
  - Initialisierung und Cleanup

Reale Objekte in C++ abbilden

- Einfaches Beispiel: $x \in \mathbb{R}^2$
- Eigenschaften:
  - x-Koordinate, y-Koordinate
  - Betrag, Winkel
- gespeicherte Daten vs. Eigenschaften
  - Eine Repräsentation zum speichern aussuchen
  - Andere Eigenschaften bei Bedarf ausrechnen
- Operationen
  - Verschieben
  - Rotieren
  - Spiegeln
  - ...

```
class Point {
public:
  double x;
  double y;
};
```

Klassen I

- C++ erlaubt die Definition von Klassen
- Eine Klasse beschreibt eine bestimmte Art von Objekt
- Alle Objekte einer Klasse sind einheitlich (Speicherbedarf etc.)
- Klassen dürfen nur in globalem Scope, in Namespaces oder in anderen Klassen definiert werden, nicht in Funktionen

```
class Point {
public:
  double x;
  double y;
};

int main() {
  Point p;
  p.x = 1.;
  p.y = 2.;
  std::cout << p.x << std::endl;
}
```

Klassen II

- Klassen beginnen mit dem keyword **`class`**, gefolgt von einem Scope, gefolgt von einem Semikolon
- Eine Klasse kann member variables enthalten
- Eine Klasse kann member functions enthalten

```
class Point {
public:
  double x;
  double y;

  double norm() {
    return std::sqrt(x*x + y*y);
  }

  void scale(double factor) {
    x *= factor;
    y *= factor;
  }
};
```

Member Functions

- muss man auf einer Instanz aufrufen
- erhalten einen impliziten ersten Parameter **this**, der die Instanz repräsentiert, für die die Methode aufgerufen wurde
- können auf die Member-Variablen der Instanz zugreifen

```
class Point {
public:
  double x;
  double y;

  double norm() {
    return std::sqrt(x*x + y*y);
  }

  void scale(double factor) {
    x *= factor;
    y *= factor;
  }
};
```

Kapselung

Klassen können die Sichtbarkeit von enthaltenen Variablen und Funktionen kontrollieren:

```
class Polygon {
// not visible outside Polygon
private:
  std::vector<Point> _corners;
// only visible to Polygon and classes that inherit from it
protected:
  const Point& corner(int i) const;
// visible to everyone
public:
  double area() const;
  void rotate(double angle);
};
```

- Die Standard-Sichtbarkeit in **`class`** ist **`private`**.
- Es ist sinnvoll, für private Member ein Namensschema einzuführen (z.B.: beginnen mit Unterstrich)

Kapselung — Richtlinien

- Alle Member-Variablen **`private`**
- Wenn externer Zugriff auf private Variablen erforderlich ist: Accessor-Methoden

```
class Complex {
  double _real, _imaginary;
public:
  double real() const { // or getReal()
    return _real;
  }

  void setReal(double v) {
    _real = v;
  }
...};
```

- In Member-Funktionen direkt auf private Variablen / Funktionen zugreifen!
- Accessor-Methoden können das Einhalten von *Invarianten* sicherstellen

const und Klassen

```
const double x = 2.0;
std::cout << x << std::endl; // ok, x wird nur gelesen
x = x + 2; // Compile-Fehler

const Complex c(1.0,2.0);
std::cout << x.real() << std::endl; // ????
c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer **const** Instanz

const und Klassen

```
const double x = 2.0;
std::cout << x << std::endl; // ok, x wird nur gelesen
x = x + 2; // Compile-Fehler

const Complex c(1.0,2.0);
std::cout << x.real() << std::endl; // ????
c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer **const** Instanz

- Woher weiß der Compiler, dass die Methode die Instanz nicht verändert?
- Lösung: Funktionssignatur so verändern, dass die Instanz (und alle Member) in der Funktion **const** sind:

```
double real() const { // this makes instance const
  return _real;       // cannot modify _real here
}
```

Initialisierung und Cleanup

- Objekte müssen vor Verwendung initialisiert werden (Speicher allokieren, Dateien öffnen etc.) und danach Ressourcen wieder freigeben.
- C++ macht hier strikte Garantien:
  - Für jedes Objekt wird ein Konstruktor aufgerufen, bevor der Programmierer Zugriff bekommt.
  - Das gilt auch für Objekte, die Member von anderen Objekten sind, und Basisklassen (siehe Vererbung).
  - Für jedes Objekt, dessen Konstruktor erfolgreich beendet wurde, wird ein Destruktor aufgerufen, bevor das Objekt aufhört zu existieren.
- Ein Objekt hört auf zu existieren, wenn
  - die Umgebung endet, in dem die Variable angelegt wurde (für normale Variablen)
  - explizit `delete` aufgerufen wird (für Pointer)
- Strengere Garantien als viele andere Sprachen.

Konstruktor


```
class Triangle : public Shape {
  Point _x1, _x2, _x3;
public:
  Triangle(const Point& x1, const Point& x2, const Point& x3)
    : Shape(), _x1(x1), _x2(x2), _x3(x3)
  {}
};
```

- Konstruktoren sind Methoden, die genauso heißen wie die Klasse und keinen Rückgabewert haben.
- Es kann mehrere Konstruktoren mit unterschiedlichen Argumenten geben.
- Vor dem Body des Konstruktors kommt die constructor initializer list:
  - Liste von Konstruktor-Aufrufen für Basisklassen und Member-Variablen
  - Wenn Basisklassen oder Variablen hier nicht aufgeführt werden, wird deren Default-Konstruktor (ohne Argumente) aufgerufen.
  - Variablen immer hier initialisieren, nicht im Body!

Destruktor

```
class Pointer {
  double* _p;
public:
  Pointer(double v)
    : _p(new double(v))
  {}

  ~Pointer()
  {
    delete _p;
  }
};
```

- Destruktoren heißen wie die Klasse mit vorgestellter Tilde "~".
- Destruktoren haben nie Argumente ⇒ es gibt nur einen pro Klasse.
- Cleanup-Aufgaben: Speicher freigeben (bei Pointern), Dateien schließen, Netzwerkverbindungen schließen, ...
- Muss man nur definieren, wenn tatsächlich eine dieser Aufgaben erfüllt werden muss

Default-Konstruktor

Der Default-Konstruktor ist der Konstruktor ohne Argumente:

```
class Empty {
public:
  Empty()
  {}
};
```

- Wenn eine Klasse keinen Konstruktor definiert, erzeugt der Compiler einen Default-Konstruktor.
- Ansonsten muss man ihn von Hand schreiben, wenn man ihn braucht.
- Der Compiler erzeugt auch keinen Default-Konstruktor wenn eine der Member-Variablen oder eine Basisklasse keinen Default-Konstruktor hat (entweder von Hand geschrieben oder default).

Objekte kopieren

- Um Objekte kopieren zu können, muss der Compiler wissen, wie er das machen soll
- Objekt beim Anlegen kopieren:
  Copy Constructor

```
Point(const Point& other)
  : _x(other._x), _y(other._y)
{}
```

- Neuen Wert in existierendes Objekt kopieren:
  Copy Assignment Operator

```
Point& operator=(const Point& other) {
  _x = other._x;
  _y = other._y;
  return *this;
}
```

- Wenn alle Member-Variablen kopierbar sind und wir kein spezielles Verhalten benötigen, kann der Compiler die Funktionen automatisch erzeugen

Ressourcenverwaltung

Programme müssen alle Ressourcen (Speicher etc.), die sie allokieren, auch wieder freigeben (sonst Bugs)!
Methoden:

Manuell Irgendwo Speicher organisieren und von Hand überlegen, wann man ihn nicht mehr braucht

- aufwendig
- fehleranfällig

Garbage Collection Speicher wird speziell markiert, in periodischen Abständen wird im Hintergrund unbenutzter Speicher gesucht und freigegeben

- komfortabel
- kann zu Programm-Rucklern führen
- Funktioniert nicht für andere Ressourcen (Dateien etc.)

RAII Die C++-Lösung

Resource Acquisition is Initialization (RAII)

C++ verwaltet Ressourcen mit dem RAII-Idiom:

- Klasse, die genau eine Ressource kapselt
- Ressource wird im Konstruktor allokiert
- Ressource wird im Destruktor freigegeben
- C++ garantiert, dass der Destruktor aufgerufen wird, falls der Konstruktor erfolgreich beendet wurde.
- Funktioniert für beliebige Arten von Ressourcen
- Der Programmierer muss die RAII-Klasse bewusst verwenden.
- Diverse Implementierungen in der Standardbibliothek:
  - Speicher: `std::vector`, `std::map`, `std::unique_ptr`, ...
  - Dateien: `std::fstream`
  - Locks: `std::lock_guard`, ...

Eigentlich wäre DIRR (Destruction is Resource Release) ein besserer Name gewesen...

RAII: Beispiel

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main(int argc, char** argv)
{
  {
    std::ofstream outfile("test.txt");
    outfile << "Hello, World" << std::endl;
  } // file gets flushed and closed here

  std::ifstream infile("test.txt");
  std::string line;
  std::getline(infile,line);
  std::cout << line << std::endl;
  return 0;
}
```

Klassen in Headerdateien

- Wenn man Klassen in Headerdateien deklariert, schreibt man die Klasse selbst in die Headerdatei.
- Funktionen werden wie üblich nur deklariert (jetzt innerhalb der Klasse).
- In der Implementierung wird die Klasse nicht erneut deklariert.
- Die Implementierung enthält nur noch Definitionen der Member-Funktionen.
- Um anzuzeigen, dass eine Funktion Member einer Klasse ist, wird dem Funktionsnamen der Klassenname gefolgt von `::` vorangestellt:

```
double Point::x() const {
  return _x;
}
```

- Die Signatur der Funktion muss exakt mit dem Header übereinstimmen, inklusive des möglicherweise angehängten **`const`**!

Klassen in Headerdateien: Beispiel I

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Headerdatei `point.hh`

```
#ifndef POINT_HH
#define POINT_HH

class Point {
  double _x;
  double _y;

public:
  // Function Declarations
  Point(double x, double y);

  double x() const;
  double y() const;

  void scale(double factor);
};

#endif // POINT_HH
```

Klassen in Headerdateien: Beispiel II

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Implementierung `point.cc`

```
#include <point.hh>

Point::Point(double x, double y)
  : _x(x), _y(y)
{}

double Point::x() const {
  return _x;
}

double Point::y() const {
  return _y;
}

void Point::scale(double factor) {
  _x *= factor;
  _y *= factor;
}
```

Objekt Orientiertes Programmieren - Zusammenfassung

- Klassen und Objekte bündeln Daten und zugehörige Methoden
  - Bessere Abstraktion von abgebildetem Verhalten
  - Objekte verwalten ihren Zustand selbst $\rightarrow$ geringere Fehleranfälligkeit
  - Zugriff auf Daten kann durch Kapselung eingeschränkt werden um z.B. invariante Variablen zu implementieren
  - Ressourcenverwaltung nach dem RAII-Idiom
  - Eigenschaften und Verhalten von Klassen können Modular mittels Komposition und Vererbung erweitert und spezialisiert werden
- Deklaration von Klassen gehören bevorzugt in Header-Files und Definition in Source-Files

Worum geht es?

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

Worum geht es?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
int_vector vec;
Point_list list;
...
// reverse order of entries
reverse_double_array(array);
reverse_int_vector (vec);
reverse_Point_list(list);
```

Worum geht es?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
int_vector vec;
Point_list list;
...
// reverse order of entries
reverse_double_array(array);
reverse_int_vector (vec);
reverse_Point_list(list);
```

Implementierung der Klassen / Funktionen jeweils fast identisch

Worum geht es?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Datenstrukturen für mehrere Typen

```
std::vector<int> v1;
std::vector<Point> v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
std::vector<int> vec;
std::list<Point> list;
...
// reverse order of entries
std::reverse(begin(array),end(array));
std::reverse(begin(vec),end(vec));
std::reverse(begin(list),end(list));
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::vector<int> v1;
std::vector<Point> v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
std::vector<int> vec;
std::list<Point> list;
...
// reverse order of entries
std::reverse(begin(array),end(array));
std::reverse(begin(vec),end(vec));
std::reverse(begin(list),end(list));
```

DRY-Prinzip: Don't repeat yourself

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- geringerer Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu lesbarerem Code

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- geringerer Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu lesbarerem Code

ABER

Funktionalität spezialisieren falls nötig

- Zusatzfunktionen
- Workarounds manchmal nötig
- Performance beachten

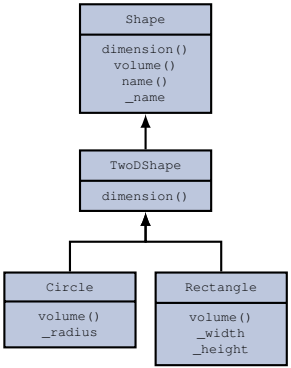
Konzepte

Standard-Konzepte für Code Reuse:

- Objektorientierte Programmierung
  - Komposition
    - Komplexe Objekte aus einfachen zusammensetzen
    - Konstruktives Prinzip
    - Bausteine als unveränderliche *black boxes*
  - Vererbung
    - Gemeinsame Funktionalität in Basisklasse
    - Abgeleitete Klassen können Funktionalität überschreiben
    - Optional: Laufzeit-Polymorphie
- Templates
  - Klassen und Funktionen, bei denen man zur Compilezeit *Typen* als Parameter angeben kann.
  - Template und Parameter-Typen nur schwach gekoppelt
  - Compilezeit-Polymorphie
  - Alle Informationen zur Compilezeit bekannt $\Rightarrow$ optimaler Code

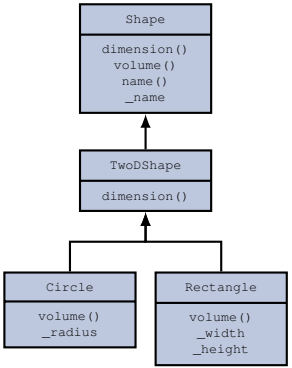
Vererbung

- Klassen können von anderen Klassen erben.
- Wichtigste Regel: is-a
  Is a circle a shape?
- Abgeleitete Klasse enthält alle Variablen und Methoden der Basisklasse.
- Methoden können überschrieben werden.
- Variablen vom Typ der Basisklasse können Objekte von abgeleiteten Klassen zugewiesen werden.
- Erweitern der Basisklasse um zusätzliche Funktionalität.



Vererbung

- Klassen können von anderen Klassen erben.
- Wichtigste Regel: is-a
  Is a circle a shape?
- Abgeleitete Klasse enthält alle Variablen und Methoden der Basisklasse.
- Methoden können überschrieben werden.
- Variablen vom Typ der Basisklasse können Objekte von abgeleiteten Klassen zugewiesen werden.
- Erweitern der Basisklasse um zusätzliche Funktionalität.



dazu später mehr

Templates: Motivation

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Beobachtung

Oft identischer Code für unterschiedliche Typen:

```
int max(int a, int b) {
  return a > b ? a : b;
}

double max(double a, double b) {
  return a > b ? a : b;
}
```

Templates: Motivation

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Beobachtung

Oft identischer Code für unterschiedliche Typen:

```
int max(int a, int b) {
  return a > b ? a : b;
}

double max(double a, double b) {
  return a > b ? a : b;
}
```

Idee

Vorlage mit Typ als Parameter:

```
SOMETYPE max(SOMETYPE a, SOMETYPE b) {
  return a > b ? a : b;
}
```

Templates: Umsetzung

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Frage

Wie Version für `int`, `double`, ... erzeugen?

- Externes Programm / Präprozessor
  - (Keine Sprachunterstützung nötig)
  - Namensgebung der Varianten?
  - Welche Varianten werden benötigt?
- Compiler (Templates)
  - Automatische Generierung aller benötigten Varianten
  - Keine unterschiedlichen Namen nötig
  - Neue Syntax erforderlich

Klassentemplates

- Syntax:

```
template<typename OneType, typename T2, int size, ...>
class MyTemplate
{
  // Parameters work like normal types and constants
  // within template
  std::array<OneType,size> _var1;
  void foo(const T2& t2);
};
```

- Typ-Parameter: Statt `typename` auch `class` erlaubt:

```
template<class T> class MyTemplate;
```

- Wert-Parameter
  - Erlaubte Typen: Eingebaute Integer (`int`, `long`, `bool`, ...).
  - Beim Verwenden des Templates müssen Werte zur Compile-Zeit bekannt sein:

```
MyTemplate<int,double,3> mt1; // ok
int size = 3; // value only known at runtime
MyTemplate<int,double,size> mt1; // compile error
```

Funktionstemplates

- Syntax:

```
template<typename T1, typename T2, ...>
T2 myFunction(const T1& t1, const T2& t2) {
  return t1.size() + t2.size();
}
```

- Aufruf:

```
std::vector<int> v1; std::vector<double> v2;
myFunction<std::vector<int>,std::vector<double>>(v1,v2);
```

- Compiler kann Template-Argumente von Laufzeit-Argumenten ableiten:

```
myFunction(v1,v2); // identical to above
```

- Wichtig: Compiler darf nie mehr als ein gültiges Template finden!
- `using namespace std;` gefährlich wegen vieler enthaltener Templates.

Template-Instantiierung

- Templates werden nur auf grundlegende Syntaxfehler geprüft, nicht kompiliert (erscheinen nicht in `.o`-Datei).
- Compiler erzeugt Template-Instanz bei Benutzung:

```
std::vector<int> vi; // Erzeugt Code für std::vector<int>
std::vector<double> vi; // Erzeugt Code für std::vector<double>
```

- Instantiierung eines Template:
  - Für einen kompletten Satz von Template-Argumenten.
  - Benötigt Zugriff auf Template-Definition.
  - Verschiedene Instantiierungen sind für C++ verschiedene Typen.
  - Kann zu Compile-Fehlern führen.
- Templates werden in jeder Translation Unit (`.cc`-Datei) einzeln instantiiert
  $\Rightarrow$ Erhöhter Compile-Aufwand
- Implementierung muß beim Instantiieren sichtbar sein!
  $\Rightarrow$ Gesamter Code in Header, keine `.cc`-Datei

Template-Instantiierung

- Templates werden nur auf grundlegende Syntaxfehler geprüft, nicht kompiliert (erscheinen nicht in `.o`-Datei).
- Compiler erzeugt Template-Instanz bei Benutzung:

```
std::vector<int> vi; // Erzeugt Code für std::vector<int>
std::vector<double> vi; // Erzeugt Code für std::vector<double>
```

- Instantiierung eines Template:
  - Für einen kompletten Satz von Template-Argumenten.
  - Benötigt Zugriff auf Template-Definition.
  - Verschiedene Instantiierungen sind für C++ verschiedene Typen.
  - Kann zu Compile-Fehlern führen.
- Templates werden in jeder Translation Unit (`.cc`-Datei) einzeln instantiiert
  $\Rightarrow$ Erhöhter Compile-Aufwand
- Implementierung muß beim Instantiieren sichtbar sein!
  $\Rightarrow$ **Gesamter Code in Header, keine `.cc`-Datei**

Template-Spezialisierung

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Es ist möglich Templates für bestimmte Template-Parameter Kombinationen zu überschreiben und somit zu spezialisieren.
- Beispiel von https://en.cppreference.com/w/cpp/language/template_specialization:

```
template<typename T>   // primary template
struct is_void : std::false_type {};
template<>  // explicit specialization for T = void
struct is_void<void> : std::true_type {};
int main() {
        // for any type T other than void, the
        // class is derived from false_type
        std::cout << is_void<char>::value << '\n';
        // but when T is void, the class is derived
        // from true_type
        std::cout << is_void<void>::value << '\n';
}
```

Concepts

- Templates akzeptieren prinzipiell jeden Typ (*duck typing*)
- Impliziter Vertrag zwischen Template und Argumenten:

```
template<typename T>
int size(const T& t) {
  return t.size();
}
```

- Argument muß Methode `int size() const` besitzen.
- In der Standard-Library Anforderungen oft in Concepts zusammengefasst:
  Copy-Constructible hat Copy-Konstruktor
  Default-Constructible hat Default-Konstruktor
  Sequence Container verhält sich wie `vector` etc.
  ...
- Wird nicht explizit geprüft, sondern führt bei Verwendung fehlender Funktionen zu schwer lesbaren Compilefehlern.
  $\Rightarrow$ Concepts als Sprachfeature in C++20

Typedefs / Aliases

- Neuen Namen für existierenden Typ vergeben:

```
typedef oldtype newtype; // C-compatible syntax
using newtype = oldtype; // new syntax (more readable)
```

- Oft in Template-Kontext verwendet:

```
template<typename T>
struct Vector {
  using Element = T;
};
...
using IntVector = Vector<int>;
IntVector::Element e = 2; // same as int e = 2;
```

Type Aliases in Templates

- Beim Parsen von geschachtelten Namen in Templates weiß der Compiler nicht automatisch, ob es sich um einen Typ oder eine Member-Variable handelt.
- Standardmässig nimmt der Compiler an, dass eine Member-Variable vorliegt.
- Wenn man einen geschachtelten Typ in einer Template verwenden will, muss man **typename** davor schreiben:

```
template<typename V>
typename V::value_type add(const V& vec) {
  // compile error in next line without "typename"
  typename V::value_type sum = 0;
  for (int i = 0 ; i < v.size() ; ++i)
    sum += v[i];
  return sum;
}
```

Keyword `auto`

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Zugriff auf type aliases in Template-Parametern oft umständlich:

```
typename T1::ScalarProduct::NormType s;
s = t1.scalarProduct().norm();
```

- `auto` *rät* Variablentypen nach gleichen Regeln wie Template-Instantiierung:

```
auto S = t1.scalarProduct().norm();
```

  - Typ deduziert aus Rückgabewert.
  - Standardmäßig immer value type (kopiert Rückgabewert).
  - Nach Bedarf mit `&` und `const` qualifizieren.
  - Kann auch für Rückgabewert von Funktionen verwendet werden.

Keyword `auto`: Beispiel

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
template<typename V>
// let the compiler deduce the return type
auto sum(const V& v)
{
  // create a variable with the type of the elements
  // in the container
  auto sum = v[0];
  // set it to zero
  sum = 0;
  // sum over all entries
  for (auto e : v)
    sum += e;
  return sum;
}
```

Templates: Beispiel


```
class Rectangle {
  ...
  double volume() { return _width * _height; }
};
class Circle {
  ...
  double volume() { return pi*_r*_r; }
};

template<typename Shape>
bool checkEmpty(const Shape& shape) {
  return shape.volume() == 0;
}

int main(int argc, char** argv) {
  Shape s;
  Circle c(radius);
  checkEmpty(s); // ruft Shape::volume() auf
  checkEmpty(c); // ruft Circle::volume() auf
}
```

Code Reuse - Zusammenfassung

- Das strenge Typ-System von C++ benötigt in vielen fällen spezialisierte Implementierungen
- C++ Templates ermöglichen das schreiben von Code für mehrere Daten-Typen
- Objektorientiertes Programmieren mit Komposition und Vererbung macht den Code Modular
- Die Menge Code kann durch diese Techniken signifikant verringert werden
  - Der Programmierer muss sich nicht unnötig wiederholen (Don't repeat yourself!)
  - Änderungen im Code betreffen weniger Stellen im Code
- Typedefs, Aliases (`using ...`) und auto-Keyword verringern Schreibarbeit und verbessern Lesbarkeit von Code

Iteratoren — Motivation

Ihr folgeneden Code kennt ihr schon:

```
std::vector<int> v(20);
...
for (auto& i : v)
  std::cout << i << std::endl;
```

Doch was passiert eigentlich Hinter den Kulissen?

Iteratoren — Motivation

Der Code kann ungefähr in folgendes übersetzt werden:

```
std::vector<int> v(20);
...
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  auto& i = *it;
  std::cout << *it << std::endl;
}
```

- Was passiert hier eigentlich?
- Was ist dieser `iterator`?
- Was macht `*it`?
- Was soll das Ganze?

Exkursion: Pointer

- Erinnerung: Jede Variable liegt an einer Adresse im Speicher
- Zugriff auf die Adresse mit Adressoperator `&`:

```
int i = 0;
std::cout << &i << std::endl;
```

- Variablen, die die Adresse einer anderen Variable speichern, heißen Pointer
- Pointer zeigen immer auf Variablen eines bestimmten Typs. Der Typ einer Pointervariablen ist der Typ der Zielvariablen mit angehängtem `*`:

```
int i = 0;
int* p = &i; // p is a pointer to int
```

- Pointer, die auf keine gültige Variable verweisen, sollte immer der spezielle Wert `nullptr` zugewiesen werden:

```
int* p = nullptr;
```

Arbeiten mit Pointern

- Pointern können neue Zielvariablen zugewiesen werden:

```
int i = 0, j = 2;
int* p = nullptr; // p does not point anywhere valid
p = &i;           // p now points to i
p = &j;           // p now points to j
```

- Um auf den Wert der Zielvariablen zuzugreifen, dereferenziert man den Pointer, indem man $\star$ voranstellt:

```
std::cout << *p << std::endl; // prints 2
```

- Wenn die Zielvariable eine Klasse ist, kann man mit $->$ (statt `.`) direkt auf Member der Zielvariablen zugreifen:

```
Point p{1.0,2.0};
Point* pp = &p;
std::cout << pp->x() << std::endl; // prints 2.0
```

Pointer als Handle für Speicher

- `array` und `vector` legen ihre Daten in einen zusammenhängenden Speicherbereich
- Pointer auf Speicherbereich mit Memberfunktion `data()`
- Pointer unterstützen mathematische Operationen:

```
std::vector<int> v(20);
int* data = v.data();
std::cout << *data << std::endl; // prints first entry
++data; // increase pointer by sizeof(int), now points to v[1]
data += 10; // now points to v[11]
std::cout << (data - v.data()) << std::endl; // prints 11
data -= 11; // points to v[0] again
```

- Vektor mit Pointern ausgeben:

```
int* end = v.data() + v.size(); // first invalid address
for(int* p = v.data() ; p != end ; ++p)
  std::cout << *p << std::endl;
```

Iteratoren: Verallgemeinerte Pointer

Warum sind Iteratoren nützlich?

- Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Iteratoren: Verallgemeinerte Pointer

Warum sind Iteratoren nützlich?

- Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Container stellen Iteratoren zur Verfügungen

- Iteratoren verhalten sich wie Pointer
- Typ des Iterators über geschachtelten Typ `Container::iterator` bzw. `Container::const_iterator` (erlaubt nur lesenden Zugriff auf Elemente)
- Iterator für erstes Element mit `begin()`
- Iterator hinter letztes Element mit `end()`
- Manche Container erlauben Rückwärtsdurchlauf mit `rbegin()`, `rend()`

Iteratoren: Beispiel

Ausgeben von Containern:

```
template<typename T>
void print(const T& t) {
  typename T::const_iterator end = t.end();
  for (auto it = t.begin() ; it != end ; ++it)
    std::cout << *it << std::endl;
}

std::vector<int> v(20);
print(v);

std::list<int> l;
...
print(l);
```

Iteratoren: Kategorien

- Je nach unterliegendem Container unterstützen Iteratoren nicht alle Pointer-Operationen
- Iterator-Kategorien:

`InputIterator`	Lesen von `*it` und `++it`
`OutputIterator`	Schreiben von `*it` und `++it`
`ForwardIterator`	Vollzugriff auf `*it` sowie `++it`
`BidirectionalIterator`	zusätzlich `--it`
`RandomAccessIterator`	zusätzlich `it += n`, `it -= n`

- Jeder Container gibt an, was für eine Iteratorkategorie er hat

Iteratoren: Operatoren

Je nach Kategorie werden folgende Operatoren implementiert:

```
iterator& operator=(const iterator&);
bool operator==(const iterator&) const;
bool operator!=(const iterator&) const;
bool operator<(const iterator&) const; //optional
bool operator>(const iterator&) const; //optional
bool operator<=(const iterator&) const; //optional
bool operator>=(const iterator&) const; //optional

iterator& operator++();
iterator operator++(int);
iterator& operator--();
iterator operator--(int);
iterator& operator+=(size_type);
iterator operator+(size_type) const;
friend iterator operator+(size_type, const iterator&);
iterator& operator-=(size_type);
iterator operator-(size_type) const;
difference_type operator-(iterator) const;

reference operator*() const;
pointer operator->() const;
reference operator[](size_type) const;
```

Algorithmen

- Alle Algorithmen in der Standardbibliothek arbeiten mit Iteratoren
- Manche Algorithmen haben Anforderungen an die Kategorie (z.B. `std::sort()`)
- Erlauben oft sehr klares Aufschreiben der Intention:

```
std::array<int,20> a;
...
// Replace all occurences of 3 with 7
std::replace(a.begin(),a.end(),3,7);

// Count number of entries with value 7
std::cout << std::count(a.begin(),a.end(),7);

std::vector<int> v;

// Copy array to vector, no need to resize
std::copy(a.begin(),a.end(),std::back_inserter(v));
```

- Erfordern Umgewöhnung und Kenntnis der Möglichkeiten

Zusammenfassung

- Iteratoren ermöglichen es unabhängig vom Containertyp über dessen Inhalt zu iterieren.
- Zugriff auf die Daten des Iterators erfolgt mit `*iterator`.
- Ein Iterator braucht in der Regel mindestens einen Operator um das nächste Element zu bekommen und einen Vergleichsoperator (typischerweise `++it` und `!=`).
- Je nach Iteratortyp kann vorwärts und rückwärts iteriert werden oder sogar auf ein beliebiges benachbartes Element zugegriffen werden.
- Viele Algorithmen arbeiten mit Iteratoren, damit diese Unabhängig von einem Containertyp implementiert werden können.

Programme haben Bugs

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Jeder von Ihnen hat schon einmal einen Fehler in einem eigenen Programm gesucht.

Wie sind Sie das angegangen?

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)
- Debugger
- überwachte Programmausführung (Sanitizers, Valgrind)

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)
- Debugger
- überwachte Programmausführung (Sanitizers, Valgrind)

Fehlersuche zur Compilezeit

- Diagnose durch den Compiler (Warnungen)
- auch als statische Programm-Analyse bezeichnet
- In manchen Sprachen sehr weitgehend bis zu mathematischen Korrektheitsbeweisen

Der Compiler will helfen

- Moderne Compiler haben eine Vielzahl an Warnungen
- Standardsatz mit `-Wall`
- Bei Fehlern eventuell `-Wextra`
- Manpage zum Finden einzelner Warnungen
- Warnungen lesen, verstehen und beheben!

Compilerwarnungen: Beispiel


```
#include <iostream>

int fibonacci(int i) {
  switch(i) {
    case 0: return 0;
    case 1: return 1;
    default: fibonacci(i-1) + fibonacci(i-2);
  }
}

int main() {
  std::cout << fibonacci(1) << std::endl;
  std::cout << fibonacci(2) << std::endl;
  std::cout << fibonacci(3) << std::endl;
}
```

Was macht dieses Programm?

Compilerwarnungen: Beispiel

Der Compiler kann hier helfen:

```
g++ -Wall fibonacci.cc
```

Ausgabe:

```
fibonacci.cc:5:33: warning: expression result unused [-Wunused-value]
        default: fibonacci(i-1) + fibonacci(i-2);
                 -------------- ^ --------------
fibonacci.cc:7:5: warning: control may reach end of non-void function [-Wreturn-type]
    }
    ^
2 warnings generated.
```

Annahmen - Assertions

Oft ist es hilfreich implizierte Annahmen zu überprüfen:

```
1 #include <cassert>
2
3 double sqrt(double x) {
4         assert(x > 0 && "x muss größer 0 sein!")
5
6         // ...
7 }
```

- assert(...) Sorgt dafür, dass das Programm abstürzt wenn die Bedingung in den Klammern nicht erfüllt ist.
- Ver-Undung mit Fehlernachricht sorgt für bessere Lesbarkeit.
- Assertions sind nur in Debug-Builds aktiv.

Debugger

Debugger sind mächtige Werkzeuge zur Laufzeit-Untersuchung

- Breakpoints können das Programm unterbrechen
  - beim Erreichen bestimmter Codezeilen
  - beim Aufrufen bestimmter Funktionen
  - optional mit Bedingungen (z.B. `x > 0`)
  - beim Zugriff auf Variablen (Watchpoints)
  - beim Werfen oder Fangen von Exceptions
- Im angehaltenen Programm können Informationen ausgegeben und verändert werden
  - Werte von Variablen
  - Die aktuelle Funktionshierarchie (Backtrace)
  - Arbeitsspeicher-Inhalte
  - Werte von CPU-Registern

Wichtig

Debugger benötigen zusätzliche Informationen über das Programm (Option `-g3`, CMake: Debug-Build)

Standard-Debugger

GDB ist der Standard-Debugger unter Linux

- unterstützt viele Programmiersprachen
- weitreichende Dokumentation im Internet
- viele IDEs können mit GDB zusammenarbeiten
- erlaubt Rückwärts-Debugging
- erfordert spezielle Konfiguration unter macOS

LLDB ist der Standard-Debugger unter macOS

- Teil des LLVM-Projekts, das auch clang entwickelt
- schlecht dokumentiert
- Kommandozeilen-Interface oft sehr umständlich
- versteht Quellcode sehr gut durch Interaktion mit clang

Typische Benutzung eines Debuggers

- Programm stürzt ab
  $\rightarrow$ Ausführen mit Debugger, untersuchen nach Fehlerquelle
- Programm tut nicht was es soll
  $\rightarrow$ Setzen von breakpoints an strategischen Stellen im Code
- In beiden Fällen nähert man sich idealerweise Schrittweise der Fehlerquelle.

GDB: Tutorial

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Vorbereitung (für alle Debugger)

Programm ohne Optimierung und mit Debug-Informationen kompilieren

- Kommandozeile: `-O0 -g3`
- CMake: Neues Build-Verzeichnis mit `-DCMAKE_BUILD_TYPE=Debug`

Speichern der eingegebenen Befehle zwischen Sitzungen aktivieren:

```
echo "set history save on" > ~/.gdbinit
```

Eine gute Übersicht und Vergleich der Befehler für GDB und LLDB gibt es hier:
https://lldb.llvm.org/use/map.html

GDB: Programm starten und Breakpoints

GDB für Programm `buggy` starten

```
gdb buggy
```

Breakpoint in Zeile 42 von `buggy.cc` setzen

```
break buggy.cc:42
```

Programm starten (oder neu starten, falls es läuft)

```
run [eventuelle Kommandozeilen-Argumente]
```

GDB: Programmsteuerung

Programm nach einem Breakpoint weiterlaufen lassen

```
continue
```

Nächste Zeile ausführen (nicht in Funktionen hineinspringen)

```
next
```

Nächste Zeile ausführen (in Funktionen hineinspringen)

```
step
```

Bis zum Ende der aktuellen Funktion weiterlaufen lassen

```
finish
```

GDB: Informationen ausgeben

Einen Ausdruck ausgeben (Variablen etc.)

```
print point.x
```

Alle lokalen Variablen anzeigen

```
info locals
```

Alle Funktionsargumente anzeigen

```
info args
```

Aktuellen Stacktrace anzeigen

```
backtrace # oder kurz bt
```

Der Stacktrace listet die ineinander geschachtelten Funktionsaufrufe auf, beginnend mit der aktuellen Funktion bis

GDB: Fazit

Mächtiges Werkzeug, aber nicht leicht zu bedienen

GDB: Fazit

Mächtiges Werkzeug, aber nicht leicht zu bedienen

$\Rightarrow$ Einfacher mit Hilfe einer IDE

Beispiel: QT Creator

Sanitizers

Sanitizer sind spezielle Compiler-Komponenten, die den generierten Code instrumentieren, so dass bestimmte Fehler zur Laufzeit erkannt werden:

- Address Sanitizer (`-fsanitize=address`): Erkennt ungültige Speicherzugriffe
- Undefined Behavior Sanitizer (`-fsanitize=undefined`): Erkennt Programmcode, der undefiniertes Verhalten auslöst. Nicht sehr zuverlässig.
- Thread Sanitizer (`-fsanitize=thread`): Erkennt Probleme bei der Programmierung mit mehreren Threads. Hier können mehrere CPU-Kerne gleichzeitig auf eine Variable zugreifen, was sogenannte data races erzeugt.
- In eine ähnliche Kategorie fällt das externe Tool valgrind, mit dem das Programm auf simulierter Hardware ausgeführt wird, die viele Fehler erkennt, aber durch die Simulation sehr langsam ist.

Warum Sanitizers?

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Oft kann ein Programm nach einem falschen Speicherzugriff etc. noch lange weiterlaufen.
- Viel später greift ein anderer Programmteil auf den überschriebenen Speicherbereich zu, und das Programm crasht.
- Ein Debugger kann nur sagen, wann dieser Crash passiert.
- Ein Sanitizer erkennt oft den wirklichen Grund für einen Bug.
- Manche Klassen von Fehlern treten in Debug-Builds nicht auf und können mit Debuggern nicht untersucht werden.
- Sanitizers können auch mit Optimierung verwendet werden, man muss nur die Debuginformation anschalten (`-g3`).

Weitere Informationen

Das Thema Sanitizer und Code Instrumentierung ist sehr umfangreich. Diese Vorlesung kann Ihnen daher nur die Existenz solcher Tools vermitteln.

- Weitere Informationen zu den GCC instrumentierungen finden Sie hier: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- Weitere Informationen zu Valgrind finden Sie hier: https://valgrind.org/docs/manual/manual-core.html

Zusammenfassung

Beim Programmieren wird es früher oder später zu Fehlern kommen. Um diese leichter zu finden, können folgende Tools verwendet werden:

- Compiler Warnings: `-Wall -Wextra`
- Assertions: `assert(<Condition> && "Message")`
- Debugger: gdb oder lldb
  - `gdb --args <binary> arg1 arg2 arg3`
  - `break <sourcefile>:<line>`
  - `info locals`
  - `print point.x`
  - `backtrace`
- Sanitizers:
  - `g++ -fsanitize=...`
  - `valgrind --tool=memcheck <binary>`

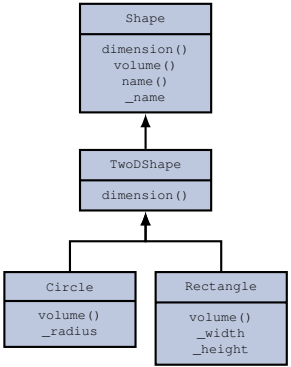
Konzepte

Standard-Konzepte für Code Reuse:

- Polymorphie/Vererbung
  - Funktionalität wird für Basisklasse geschrieben.
  - Akzeptiert auch Objekte von abgeleitetem Typ.
  - Beschränkt auf objektorientierte Programmierung.
  - Optional: Laufzeit-Polymorphie
- Templates
  - Der Typ selbst wird ein Parameter.
  - Akzeptiert jeden Typ, für den der geschriebene Code kompiliert werden kann.
  - Compilezeit-Polymorphie

Vererbung

- Klassen können von anderen Klassen erben.
- Wichtigste Regel: is-a
  Is a circle a shape?
- Abgeleitete Klasse enthält alle Variablen und Methoden der Basisklasse.
- Methoden können überschrieben werden.
- Variablen vom Typ der Basisklasse können Objekte von abgeleiteten Klassen zugewiesen werden.
- Erweitern der Basisklasse um zusätzliche Funktionalität.



UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Motivationsbeispiel:

```
class GenericAmoeba {...};

class CautiousAmoeba : public virtual GenericAmoeba {...};

class ControllableAmoeba : public virtual GenericAmoeba{...};

class RandomAmoeba : public virtual GenericAmoeba {...};

class RandomControllableAmoeba :
        public RandomAmoeba, public ControllableAmoeba {...};
```

Verwendung von Klassenhierarchien

- Referenzen und Pointer auf Basisklassen funktionieren auch mit abgeleiteten Klassen:

```
Circle c(...);
Shape& s_ref = c;
```

- Beim Kopieren von Objekten werden nur die enthaltenen Daten der Basisklasse kopiert, Informationen aus abgeleiteten Klassen gehen verloren:

```
// only copies member variable _name
Shape s_copy = c;
```

- Eine Referenz auf die Basisklasse hat nur Zugang zu den Methoden und Variablen der Basis:

```
s_ref._name;  // ok
s_ref._radius // compile error
```

- Aufgerufene Funktionen sind immer aus der Basisklasse:

```
c.volume()      // calls Circle::volume()
```

Dynamische Polymorphie

- Idee: Beim Aufruf einer Methode die Implementierung aus der abgeleiteten Klasse verwenden:

```
Circle c(...);
Shape& s_ref = c;
s_ref.volume(); // calls Circle::volume()
```

- Funktioniert mit **`virtual`** Funktionen:

```
class Shape {
  virtual double volume() const;
  // always make destructor virtual as well!
  virtual ~Shape();
};
```

- Methode ist dadurch auch in allen abgeleiteten Klassen **`virtual`**.
- Funktioniert nur mit Pointern / Referenzen:

```
s_ref.volume(); // calls Circle::volume()
Shape s_copy = c;
s_copy.volume(); // calls Shape::volume()
```

Dynamische Polymorphie: Pitfalls

- Keyword **virtual** ist in abgeleiteten Klassen implizit, aber die Redeklaration ist erlaubt.
- Methoden-Signatur in abgeleiteten Klassen muss exakt identisch sein, inklusive **const**-Deklarationen:

```
class Circle {
  // does NOT override the volume() method in Shape!
  // (we forgot the const)
  virtual double volume();
}
```

- Besser: **override**, um Tippfehler zu vermeiden:

```
class Circle {
  double volume() const override; // ok
  // compile error: no virtual function defined in base class
  double volume() override;
```

- Immer auch den Destruktor **virtual** machen, ansonsten oft Speicherlücken und ähnliche Probleme!

Abstrakte Klassen

- Es gibt auch die Möglichkeit abstrakte Klassen, welche nicht instantiiert werden können zu implementieren.

```
class Abstract {
  virtual void doSomething() = 0; // pure virtual function
};
```

- Abstrakte Klassen werden typischerweise zur Definition von Interfaces genutzt.
- Eine Klasse wird abstrakt durch Definition einer rein virtuellen Funktion.
- Die Ableitende Klasse muss die rein virtuelle Funktion implementieren!

```
class Derived {
  void doSomething() override { ... }
};
```

Ableitung von mehreren Klassen

- Eine Klasse kann auch von mehreren Basisklassen ableiten
- Hierbei kann das virtual Keyword verhindern dass die Basisklasse mehrfach in der abgeleiteten Klasse vorkommt.

```
class Base {
public:
        int n;
        Base(int x) : n(x) {}
};
class D1 : virtual Base { public: D1() : Base(1) {} };
// virtual prevents multiple instaces of Base in inheritance tree
class D2 : virtual Base { public: D2() : Base(2) {} };
class DMulti : D1, D2 { public: DMulti() : Base(3), D1(), D2() };
```

- In der abgeleiteten Klasse müssen alle Konstruktoren der Basisklassen aufgerufen werden ansonsten wird der Default-Konstruktor wendet (wenn möglich).
- Die Klassen von denen abgeleitet wird müssen keine gemeinsame Basisklasse haben.

Speichern von polymorphen Objekten

- Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt:
  `sizeof(Circle) != sizeof(Rectangle)`
- Container brauchen Objekte fixer Grösse
- Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Speichern von polymorphen Objekten

- Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt:
  `sizeof(Circle) != sizeof(Rectangle)`
- Container brauchen Objekte fixer Grösse
- Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Lösung: Dynamische Speicherverwaltung

- Wir speichern eine Liste von Pointern auf Objekte und lassen uns dynamischen Speicher für das eigentliche Objekt geben
- Man spricht davon, dass das Objekt auf dem Heap angelegt wird, normale Variablen liegen auf dem Stack
- Objekte auf dem Heap werden NICHT automatisch aufgeräumt, wenn das aktuelle Scope endet
- Bei manueller Verwaltung: Speicher geht eventuell verloren
- Daher: smart pointer verwenden!

Smart Pointers

- Ein Smart Pointer reserviert Speicher für ein Objekt auf dem Heap und räumt das Objekt auf, wenn es nicht mehr verwendet wird.
- `unique_ptr` erzeugt das neue Objekt beim Anlegen und gibt es frei, sobald die Pointer-Variable aufhört zu existieren:

```
#include <memory>

std::unique_ptr<int> foo(int i) {
  return std::make_unique<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  return *p + b;
} // memory gets freed here
```

- `unique_ptr` kann nur verschoben werden, nicht kopiert

Anwendungsbeispiel mit unique pointer

- Bei Containern unbedingt mit inplace mit `emplace_back` anlegen.
- `push_back` macht eine Kopie, was bei unique Pointern nicht erlaubt ist.

```
vector<unique_ptr<Base>> v;
v.emplace_back(make_unique<Base>(Base()));
v.emplace_back(make_unique<Derived1>(Derived1()));
v.emplace_back(make_unique<Derived2>(Derived2()));

for (auto& br : v)
  br->foo();
```

Smart Pointers für geteilte Objekte

- Oft ist es nicht möglich, einen eindeutigen Eigentümer für ein Objekt festzulegen.
- Hierfür gibt es `shared_ptr`.
- Mehrere `shared_ptr` können auf das gleiche Objekt zeigen.
- Das Objekt wird genau dann freigegeben, wenn der letzte `shared_ptr` auf das Objekt zerstört wird.
- Wichtig: `shared_ptr` immer nur mit `make_shared` anlegen oder aus anderen `shared_ptr`n kopieren!

```cpp
#include <memory>

std::shared_ptr<int> foo(int i) {
  return std::make_shared<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  auto p2 = p;
  return *p + *p2 + b;
} // memory gets freed here
```

Zusammenfassung

- Programm entscheidet zur Laufzeit, welche Methode ausgeführt wird.
  - Vorteil: Hohe Flexibilität (die gleiche Funktion kann zur Laufzeit für zwei Objekte unterschiedlichen Typs jeweils die richtige Methode aufrufen).
  - Nachteil: Laufzeit-Overhead (die richtige Methode muß zur Laufzeit identifiziert werden).
- Erfordert Planung und Disziplin beim Programm-Design:
  - Gemeinsame Hierarchie für alle Klassen.
  - Gemeinsame Funktionalität muß in Basisklasse vorgesehen sein (`virtual`-Deklarationen).
  - Vorhandene Klassen (z.B. aus Standard Library) nicht integrierbar.

Motivation (I)


```
class Vector {
public:
  Vector();
  Vector(double x, double y);
  Vector(const Vector& v);
  void add(const Vector& v);
  void subtract(const Vector& v);
  void scale(double s);
}
```

Gegeben $\mathbf{u}, \mathbf{v} \in \mathbb{R}^2, a \in \mathbb{R}$: Berechne $\mathbf{u} = \mathbf{u} + a \cdot \mathbf{v}$

```
Vector av(v);
av.scale(a);
u.add(av);
```

Die deutlich schlechtere Lesbarkeit im Vergleich zu `u = u + a * v;`

Motivation (II)

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
class Vector {
public:
  double x() const;
  double y() const;
  ...
  void print(std::ostream& os) const {
    os <<  "(" << x() <<  ", " << y() <<  ")";
  }
};

std::cout << "Result:  ";
u.print(std::cout);
std::cout << std::endl;
```

Etwas wie `std::cout << u;` wäre doch praktisch, oder?

Lösung: Operator Overloading

In C++ können fast alle Operatoren überladen werden, wenn mindestens ein Operand eine Klasse oder eine Enumeration ist.

- Überladene Operatoren definiert durch spezielle Funktionen mit festem Namensschema: `operator?()`, wobei `?` durch den zu überladenden Operator zu ersetzen ist
- Verschiedene Typen von Operatoren:
  - Unäre Operatoren: ein Operand, z.B. `-x`, `!x`
  - Binäre Operatoren: zwei Operanden, z.B. `a + b`, `a << b`
  - Spezielle Operatoren: z.B. `a[b]`, `a(b,c)`
- Weitgehend vollständige Auflistung aller Operatoren sowie weitere Informationen unter https://en.cppreference.com/w/cpp/language/operators
- Operatorfunktionen entweder als Memberfunktionen von Klassen oder als freistehende Funktionen
- Bei Operatoraufruf keine Namespace-Angabe möglich $\Rightarrow$ freistehende Funktionen immer in Namespace des eigenen Objekts, damit der Compiler sie findet (ADL: argument-dependent lookup)

Unäre Operatoren

- Member-Funktion:

```
class Vector {
public:
  Vector operator-() const {
    return Vector(-x(),-y());
  }
};
```

- Freistehende Funktion:

```
Vector operator-(const Vector& v) {
  return Vector(-v.x(),-v.y())
}
```

Binäre Operatoren

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Member-Funktion:

```
class Vector {
public:
  Vector operator+(const Vector& b) const {
    return Vector(x() + b.x(),y() + b.y());
  }
};
```

- Freistehende Funktion:

```
Vector operator+(const Vector& a, const Vector& b) {
  return Vector(a.x() + b.x(),a.y() + b.y());
}
```

Binäre Operatoren mit unterschiedlichen Typen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Reihenfolge der Argumente ist wichtig!
- Member-Funktion:

```
class Vector {
  public:
  Vector operator*(double s) const {
    return Vector(s * x(),s * y());
  }
};
```

  - Objekt ist immer der linke Operand!
  - Funktioniert nur für `v * s`, nicht für `s * v`

Binäre Operatoren mit unterschiedlichen Typen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Reihenfolge der Argumente ist wichtig!
- Member-Funktion:

```
class Vector {
  public:
  Vector operator*(double s) const {
    return Vector(s * x(),s * y());
  }
};
```

  - Objekt ist immer der linke Operand!
  - Funktioniert nur für `v * s`, nicht für `s * v`
- Zwei freistehende Funktionen:

```
Vector operator*(const Vector& v, double s) {
  return Vector(s * v.x(),s * v.y();
}

Vector operator*(double s, const Vector& v) {
  return v * s; // forward to other implementation
}
```

Freistehende Funktionen als `friend`s

Freistehende Operator-Funktionen oft erforderlich, aber

- haben keinen Zugriff auf private Variablen / Methoden
- nicht innerhalb der Klassendeklaration, schwer zu finden

Freistehende Funktionen als `friend`s

Freistehende Operator-Funktionen oft erforderlich, aber

- haben keinen Zugriff auf private Variablen / Methoden
- nicht innerhalb der Klassendeklaration, schwer zu finden

Freistehende Funktionen können mit einer Klasse befreundet sein

- Deklaration (und möglicherweise Definition) innerhalb der Klasse
- Kennzeichnung durch Voranstellen von `friend`
- Voller Zugriff auf alle privaten Variablen und Methoden
- Keine Member-Funktion!

```
class Vector {
public:
...
  friend Vector operator*(double s, const Vector& v) {
    return Vector(s * v.x(),s * v.y());
  }
};
```

Klassen mit Unterstützung für Ein- / Ausgabe

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Um Eingabe und Ausgabe mit C++-Streams zu unterstützen, muss eine Klasse die passenden Operatoren überladen:

```
class Vector {
public:
...
  friend std::ostream& operator<<(
    std::ostream& os, const Vector& v) {
    os << "(" << x() << ", " << y() << ")";
    return os;
  }

  friend std::istream& operator>>(
    std::istream& is, Vector& v) {
    ...
    return is;
  }

};
```

- Syntax sowie wie hier

Funktoren: Funktionen mit Gedächtnis

Manchmal braucht man eine Funktion, die sich Informationen zwischen den Aufrufen merken kann:

- Eine Funktion, die zu einem Argument immer eine feste, aber zur Laufzeit bestimmte Zahl hinzuaddiert
- Eine Funktion, die weiß, wie oft sie schon aufgerufen wurde

Funktoren: Funktionen mit Gedächtnis

Manchmal braucht man eine Funktion, die sich Informationen zwischen den Aufrufen merken kann:

- Eine Funktion, die zu einem Argument immer eine feste, aber zur Laufzeit bestimmte Zahl hinzuaddiert
- Eine Funktion, die weiß, wie oft sie schon aufgerufen wurde

Lösung: Funktoren: Objekte, die man wie eine Funktion aufrufen kann.

```
class Funktor {
public:

  T operator()(A a, B b, C c) const {
    ...
  }
};
```

Funktoren: Beispiel

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

```
template<typename T>
class add {
  T _number;
  int _calls = 0;
public:

  add(T number)
    : _number(number)
  {}

  template<typename U>
  auto operator()(const U& u) const {
    ++_calls;
    return u + _number;
  }
};
```

Funktoren in der Standardbibliothek

Viele Algorithmen in der STL akzeptieren Funktoren:

- `std::sort`
- `std::find_if`
- `std::copy_if`
- `std::transform`
- `std::generate`
- ...

Algorithmen können so angepasst werden:

- Sortiere Berge absteigend nach Höhe
- Kopiere alle Berge höher als 8.000m
- Extrahiere Liste von Erstbesteigern aus Liste von Bergen
- ...

STL-Funktoren: Beispiel

Im folgenden operieren wir mit folgender Klasse:

```
struct Mountain {
  std::string name;
  int height;
  int first_ascent;
  std::vector<std::string> first_ascenders;
};
```

- Die Klasse ist mit als **`struct`** definiert, alle Member sind also **`public`**.
- Aus Lesbarkeitsgründen greifen wir im folgenden direkt auf die Member-Variablen zu.

Ausserdem haben wir eine Liste von Bergen:

```
std::vector<Mountain> mountains = ...;
```

Berge sortieren

Wir können `mountains` nicht mit `std::sort()` sortieren, weil der Compiler nicht weiß, welcher Berg zuerst kommen soll.

- Wir können einen Vergleichsoperator für die Relation $<$ definieren. Dieser wird von `std::sort()` verwendet:

```
bool operator<(const Mountain& m1, const Mountain& m2) {
  return m1.name < m2.name; // Sort mountains alphabetically
}
```

- Falls wir die Berge anders sortieren wollen, können wir `std::sort()` dies explizit sagen:

```
std::sort(
  mountains.begin(),
  mountains.end(),
  sort_mountains_by_descending_height()
  );
```

Die Definition von `sort_mountains_by_descending_height` folgt auf der nächsten Folie.

Berge sortieren: Funktor

Bevor wir die Berge nach Höhe sortieren können, müssen wir ausserhalb der Funktion, in der wir sortieren wollen, den passenden Funktor definieren:

```
struct sort_mountains_by_descending_height {

  bool operator()(
    const Mountain& m1,
    const Mountain& m2
    ) const
  {
    // the functor returns whether the first
    // argument is smaller than the second
    return m1.height > m2.height;
  }

};
```

Lambda-Funktionen: Motivation

Oft wird ein Funktor nur einmal beim Anruf eines Algorithmus benötigt

- Definition als Klasse weit weg von Verwendung
- Viel “Boilerplate”

Lambda-Funktionen erlauben inline-Definition von Funktoren als Variablen

```
auto sort_height = [](auto& m1, auto& m2) {
  return m1.height > m2.height;
};
```

Lambda-Funktionen: Syntax


```
[capture-list](arg-list) -> return-type {
  body
};
```

Die Definition einer Lambda-Funktion besteht immer aus

- einer capture list in `[]`, die steuert, welche Variablen aus dem aktuellen Scope im body der Lambda-Funktion verfügbar sind.
- einer Liste von Funktions-Argumenten in `()`.
- dem eigentlichen Funktionscode in `{}`.

Meistens kann der Compiler den Rückgabetyp der Lamba-Funktion erraten, ansonsten kann er optional mit `-> return-type` angegeben werden.

Lambda-Funktionen: Implementierung

Intern sind Lambda-Funktionen bis auf Details nur eine Kurzschreibweise für die Definition eines Funktors und die Erzeugung einer Variable vom Typ des Funktors:

```
auto squared = [](double i) {
  return i * i;
};
```

ist äquivalent zu

```
// type name is neither known nor relevant
struct unknown_type {
  auto operator()(double i) const {
    return i * i;
  }
};
...
{
  auto squared = unknown_type();
}
```

Lambda-Funktionen: Capture-Spezifikation (I)

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Standardmässig kann man in einer Lambda-Funktion nicht auf die Variablen des umgebenden Scopes zugreifen.
- Um diese Variablen verfügbar zu machen, kann man sie in der Capture-Spezifikation auflisten:

Variable: `var`	Die Variable im Lambda ist eine Kopie.
Variable + `&`: `&var`	Die Variable im Lambda ist eine Referenz auf die Original-Variable.
Ein einzelnes `&`	Im Lambda verwendete Variablen sind automatisch per Referenz verfügbar.
Ein einzelnes `=`	Verwendete Variablen sind automatisch als Kopie verfügbar.

```
double a = 2.0, y = 3.0;
auto axpy = [=,&y](double x) {
  // default capture: by copy, b captured by reference
  return a * x + y;
}
a = 4.0; // does not change the lambda
y = 3.0; // changes the lambda
```

Generische Lambda-Funktionen

- Oft ist es praktisch, wenn eine Lambda-Funktion für verschiedene Typen von Argumenten funktioniert (wie eine Template-Funktion).
- Bei Verwendung von **`auto`** in der Parameter-Liste wird der **`operator`**`()` im Funktor ein Template und jeder **`auto`**-Parameter ein Template-Argument:

```
auto plus = [](auto a, auto b) {
  return a + b;
};
```

erzeugt den Funktor

```
struct unknown_plus_type {
  template<typename T1, typename T2>
  auto operator()(T1 a, T2 b) const {
    return a + b;
  }
};
```

Lambda-Funktionen: Hinweise (I)

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

- Der genaue Typ des Funktors wird vom Compiler festgelegt und kann nicht aufgeschrieben werden.
- Lambda-Funktionen kann man daher nur in einer **auto**-Variablen speichern.
- Man kann Lambda-Funktionen auch an Template-Parameter binden:

```
template<typename Functor, typename Arg>
auto call(Functor f, Arg arg)
{
  return f(arg);
}
...
call(axpy,3.0);
```

Lambda-Funktionen: Hinweise (II)

- Captures werden intern zu Member-Variablen des Funktors.
- Der Compiler generiert einen passenden Konstruktor und den zugehörigen Aufruf.
- Vorsicht mit der Lebenszeit von Variablen bei Capture by reference, wenn das Lambda von der Funktion zurückgegeben wird:

```
auto makeLamba(int add) {
  return [&](int i) {
    return i + add;
  };
} // boom!
```

In diesem Beispiel speichert die Lambda-Funktion eine Referenz auf die Variable `add`, die aber nach dem Verlassen von `makeLambda()` nicht mehr gültig ist!

Bedingtes Kopieren

- Die Funktion `std::copy_if` kopiert die Werte aus einer Iterator-Range, für die das Predicate (ein Funktor, der ein **`bool`** zurückgibt) `true` ist:

```
template<typename InIt, typename OutIt, typename Pred>
OutIt copy_if(
  InIt first, InIt last,
  OutIt out_first,
  Pred predicate);
```

- Wir wollen eine Liste mit hohen Bergen:

```
std::vector<Mountain> high;
int min_height = 5000;
std::copy_if(
  mountain.begin(),mountain.end(), // source
  std::back_inserter(high), // append copied values to high
  [=](auto& mountain) {
    return mountain.height >= min_height;
  }
);
```

Zählen von Elementen

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

- Die Funktion `std::count_if` zählt, für wie viele Elemente das Predicate `true` ist:

```
template<typename It, typename Pred>
std::size_t count_if(It first, It last, Pred predicate);
```

- Wir wollen wissen, wie viele Berge erst nach 1900 bestiegen wurden:

```
auto late_ascents = std::count_if(
  mountain.begin(),mountain.end(), // source
  [=](auto& mountain) {
    return mountain.first_ascent >= 1900;
  }
);
```

Zusammenfassung

- C++ erlaubt das Überladen von fast allen Operatoren.
- Operatoren werden als Member-Funktionen oder als Freistehende Funktionen definiert
- Freistehende Funktionen müssen ggf. 'friend' des Operanden sein.
- Klassen, welche den `operator()` überschreiben sind Funktoren, also Funktionen mit Gedächtnis (bzw. internem Zustand).
- Funktoren sind nützlich zur Definition von Prädikaten (count_if etc.) und um das Verhalten von Algorithmen zu erweitern/modifizieren (sortieren nach anderen Kriterien als dem Vergleichsoperator).
- Lambda-Funktionen sind ,,Wegwerf"-Funktoren und eine Kurzschreibweise um Funktoren zu implementieren.
- In Lambdas können die Variablen des aktuellen Scopes verfügbar gemacht werden. Diese können ganz Allgemein als Referenz (`[&]`) oder Kopie (`[=]`) verfügbar gemacht werden. Darüber hinaus können auch nur bestimmte Variables ,,gecaptured" werden.

Online Resourcen

Hier noch ein paar hilfreiche Links fürs Programmieren

- `https://cppreference.com` - C++ Referenz Dokumentation
- `https://linux.die.net/man/` - Online Man Pages
- `https://onlinegdb.com/online_c++_compiler` - Online C++ Compiler