Übung zur Vorlesung Programmieren 2
SoSe 2024

Blatt 1

IDE Setup, Hello World, Debugging

Erfordert die VL bis Kapitel	Grundlagen
Abgabe bis	23. April 2024, 16:00 Uhr

Richtlinien für Abgaben:

Lösungen, die diese Richtlinien nicht erfüllen, werden mit *0 Punkten* gewertet!

Abgaben (Code/Text) erfolgen über den Git-Server (↗) des Instituts.
Jeder in der Gruppe ist für die (pünktliche) Abgabe mitverantwortlich. Für die Abgabe gilt nicht die Commit-Zeit, sondern der Push-Zeitpunkt.

In Git ...
- wird für jedes Übungsblatt ein neuer Ordner angelegt: *Gruppe_XY/blatt1*, */blatt2*, */blatt3*, ...
- dürfen keine temporären/automatisch generierten Dateien hinzugefügt werden (→ *.gitignore*).
- wird nur auf den Server gepusht, wenn der Code im aktuellen Commit lauffähig ist.
  Nicht lauffähige lokale Commits (für Zwischenstände) sind OK und sogar empfohlen. Diese sollten allerdings mit `"WIP:"` (work in progress) am Anfang der Commit-Nachricht gekennzeichnet werden.

Es wird kein vollständiger Code-Style vorgegeben, aber ...
- innerhalb jeder Abgabe muss die Code-Formatierung konsistent sein.
- es wird sinnvoll und konsistent eingerückt.
- Variablen haben aussagekräftige Namen.
- Compiler-Warnungen im eigenen Code gelten als Fehler.

Allgemeine Infos:

Die Aufgaben und Erklärungen orientieren sich an (C++17) CMake-Projekten in der CLion IDE (↗), für Studenten kostenlos verfügbar für Windows, macOS und Linux.

Richtlinien & Infos werden bei Bedarf angepasst/erweitert. Sie gelten entsprechend je Übungsblatt.

Hinweise zu Blatt 1:

Feiertag: Für Gruppen, deren kleine Übung wegen dem Tag der Arbeit ausfallen, verschiebt sich die Präsentation von Blatt 1 um eine Woche. Das betrifft jedoch nicht die Abgabefrist.

`Graue Stellen` `in Code-Ausschnitten` können in eurer Abgabe abweichen.

Aufgabe 1

1 Punkt

> C++-Code muss, wie bei allen Compiler-Sprachen, von einem Compiler in vom jeweiligen Betriebssystem (BS) ausführbaren Code übersetzt/kompiliert werden. Dementsprechend gibt es für verschiedene BS auch verschiedene Compiler (GCC, Visual C++, CLang, ...). Statt diese direkt per Shell zu benutzen, werden IDEs genutzt, die einen entsprechenden Compiler ausführen.

Richtet IDE und Compiler zur Entwicklung von C++-Anwendungen ein. Wir empfehlen CLion (s.o.).

Aufgabe 2

1 Punkt

> CMake ermöglicht die Definition von Projekten in einem unabhängigen Format, statt dem eines speziellen Compilers oder einer speziellen IDE. So können Teams auf unterschiedlichen BS und mit unterschiedlichen Tools an einem Projekt zusammenarbeiten.

Erstellt ein neues Projekt für eine CMake-basierte C++-Anwendung mit folgenden Dateien, kompiliert es und führt es aus.

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.17)
project(hello)

set(CMAKE_CXX_STANDARD 17)

add_executable(hello main.cpp)
```

main.cpp:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello, World!" << endl;

  return 0;
}
```

Abgabe/Präsentation:

Zeigt, dass das Programm bei **jedem** Teammitglied in der IDE kompiliert und läuft.

Aufgabe 3

1 Punkt

> Zur Code-Versionierung (und Abgabe) aller Übungen wird Git verwendet. Dabei ist wichtig, dass temporäre/automatisch generierte Dateien (also Dateien, die ihr nicht selbst erstellt habt, wie z.B. der Inhalt von build-Verzeichnissen) **niemals** ins Git gehören. Statt aber bei jedem Commit darauf achten zu müssen, bietet Git sogenannte *.gitignore* Dateien, mit deren Hilfe unerwünschte Dateien von Git automatisch "ignoriert" werden. Eine Erklärung der Syntax dieser Datei gibt es z.B. hier: `https://git-scm.com/docs/gitignore`
>
> `> git status` zeigt unter *Untracked files* Dateien, die (noch) nicht in Git gesichert sind.
>
> `> git ls-files` zeigt alle in Git gesicherten Dateien an.
>
> Automatisch generierte Dateien sollten in keiner dieser Ausgaben auftauchen.

Überprüft nach der Bearbeitung von Aufgabe 2, ob Git Dateien erkennt, die automatisch generiert wurden. Falls ja, fügt diese in der Datei *.gitignore* hinzu.

Wiederholt dies für alle in eurer Gruppe verwendeten Betriebssysteme und IDEs.

Abgabe/Präsentation:

Gebt in einer Textdatei *aufgabe3.txt* die Ausgabe von `> git status` und `> git ls-files` an.

Aufgabe 4

5 Punkte

> Debugging nimmt einen erheblichen Anteil der Zeit (oft über 50%) beim Programmieren ein. Moderne Compiler und IDEs versuchen daher möglichst präzise Informationen über Fehler zu berichten. Werden mehrere Fehler gemeldet, sollte versucht werden diese der Reihe nach zu lösen, da spätere Fehlerberichte unter Umständen nur als "Folgefehler" durch einen früheren Fehler auftreten.

Um uns mit dem Debugging in C++ vertraut zu machen, wollen wir jetzt einen Vektor (ein dynamisches Array) mit den Namen aller Teammitglieder erstellen und anschließend in gekürzter Form ausgeben. Ergänzt dazu die Datei *main.cpp* wie folgt und versuche den Code erneut zu kompilieren.

main.cpp (erweitert):

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

using namespace std;

int main() {
  cout << "Hello, World!" << endl;

  cout << "Wir sind: " << team[0] << "..." << team[teamSize] << endl;
  vector<string> team = { "Paddy", "Linda" }; // Vektor mit Startwerten
  team.emplace_back("Steve"); // weiteren Wert zum Vektor hinzufügen
  team.emplace_back("Flo"); // ...
  int teamSize = team.size();

  return 0;
}
```

Scheinbar ist uns hier die Reihenfolge einiger Befehle durcheinander geraten. 😇

Abgabe/Präsentation:

1. Gebt in einer Textdatei *aufgabe4.txt* die erste Fehlermeldung des Compilers an, und zeigt daran ...
   (a) in welcher Datei der Fehler auftritt.
   (b) in welcher Zeile dieser Datei der Fehler auftritt.
   (c) den Code-Ausdruck bei dem der Fehler auftritt.
   (d) die Beschreibung des Fehlers.
2. Beschreibt in der selben Datei mit eigenen Worten, warum der Code nicht kompiliert.

Aufgabe 5

3 Punkte

> Wenn man sich nicht sicher ist, wie/wann ein Fehler im Programmverlauf entsteht, können Breakpoints ein hilfreiches Debugging-Tool sein. Sie bieten die Möglichkeit Code ab einem bestimmten Punkt schrittweise auszuführen, und währenddessen die jeweils aktuellen Werte von Variablen zu überprüfen.

Verschiebt Zeile 10 (`cout << "Wir sind...`) an das Ende der Funktion (vor `return 0;`).

Mit der korrekten Reihenfolge der Befehle sollte es jetzt wieder kompilieren — oder?
Findet mithilfe des Debuggers den verbleibenden Fehler im Code.

Abgabe/Präsentation:

1. Setzt einen Breakpoint in der letzten Zeile, von der ihr nach der Änderung noch sicher wissen könnt, dass sie korrekt funktioniert.
2. Zeigt in Debugger-Einzelschritten, wie `team` gefüllt wird, und warum `team[teamSize]` nicht den Namen des letzten Teammitglieds liefert, sondern fehlschlägt.
3. Korrigiert den Code zur Ausgabe des letzten Teammitglieds.

Gesamt: 11 Punkte