

# Programmaufteilung

Thomas Hausberger, Matthias Panny  
Ursprünglich erstellt von Sebastian Stabinger

SS2022

Wozu sollte man ein  
Programm aufteilen?

- In C und C++ kann man prinzipiell beliebig große Programme in einer Datei schreiben (das geht nicht in allen Sprachen)
- Größere Programme schreibt man aber üblicherweise auf mehrere (oft sehr viele) Dateien aufgeteilt
  - Der Linux Kernel hat aktuell z.B. knapp 60.000 Dateien mit 24.000.000 Zeilen Code

## Vorteile einer Aufteilung

- Einfacheres Zusammenarbeiten mit anderen Programmierern
- Übersichtlicher
- Die Compilezeiten verringern sich sehr

- Es ist relativ schwierig eine Datei mit mehreren Leuten gleichzeitig zu bearbeiten
- Häufig muss man für die Implementierung einer neuen Funktion das Programm **kurzzeitig in einen ungültigen Zustand bringen** (das Programm compiliert nicht mehr). Das wäre für andere Beteiligte relativ unpraktisch.
- Üblicherweise verwendet man ein **Version Control System** um die Änderungen in einem Projekt nachvollziehen zu können und mehreren Programmierern Änderungen zu erlauben. Das populärste VCS ist aktuell **git**  
<https://en.wikipedia.org/wiki/Git>

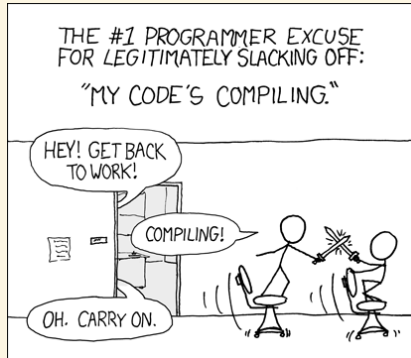
## Änderungsverlauf des git-Repositories dieser Lehrveranstaltung

Commits in master			
7892537	*	master origin/master graded exercise 7	Simon Walter Hangl 11 months
56672c5	*	graded exercise 6	Simon Walter Hangl 11 months
98db28d	*	graded sheet5	Simon Walter Hangl 11 months
8816ed4	*	Merge branch 'master' of github.com:shangl/mci	Simon Walter Hangl 11 months
\			
13d1a0a		* Sample Programs V014 Group B	Sebastian Stabing... 11 months
691cd09		* Add sample programs from lecture 12/13 group B	Sebastian Stabing... 12 months
0c6199f		* Add sample programs of group A 13/14	Sebastian Stabing... 12 months
c0b1f39		* Add health bars to sebastian example project	Sebastian Stabing... 12 months
184a66e		* Fixes unicode bug in exception slides	Sebastian Stabing... 12 months
95cbd67		* Added project for students sebastian including minimal implementation	Sebastian Stabing... 12 months
58c5f6d		* Converted .bmp tiles to a better supported format	Sebastian Stabing... 12 months
278ee2f		* Updated MCIGraphTemplate with new library	Sebastian Stabing... 12 months
4ff4d7e		* Updated MCIGraph with additional features	Sebastian Stabing... 12 months
ecb7cda	*	corrected sheets 2 and 3	Simon Walter Hangl 11 months
\			
78fc6cf	*	final projects	Simon Walter Hangl 12 months
ceada73	*	final projects	Simon Walter Hangl 12 months
bcb8526	*	started final project description	Simon Walter Hangl 12 months
658505b	*	Merge branch 'master' of github.com:shangl/mci	Simon Walter Hangl 12 months
\			
54c8302		* Adds sample programs for lecture 10 11 group A	Sebastian Stabing... 12 months
fe8be57	*	...	Simon Walter Hangl 12 months
\			

- Es ist wesentlich einfacher bestimmte Teile eines Programms zu finden wenn mehrere Dateien verwendet werden
- Wenn z.B. jede Klasse in einer eigenen Datei liegt findet man diese sehr einfach über den Klassennamen
- Mit einer modernen IDE ist das kein so großes Problem mehr wie früher
  - Es gibt z.B. üblicherweise eine eigene Klassenansicht die unabhängig von Dateien arbeitet
  - Es kommt aber immer wieder vor, dass man ohne IDE auskommen muss, oder externe Tools verwendet werden die mit einer Aufteilung auf Dateien übersichtlicher werden

# Compilezeiten

- Große Projekte können **Stunden** für einen kompletten Compilevorgang benötigen. Dadurch kann das Programmieren extrem ineffizient werden
- Wenn ein Programm auf mehrere Dateien aufgeteilt ist müssen nur die Dateien neu compiliert werden die sich **geändert** haben



Wie teilt man Programme auf  
mehrere Dateien auf?



# Das generelle Prinzip

- Wir wollen also Teile unseres Programms aus der Hauptdatei (in der die `main`-Funktion liegt) auslagern
- In C/C++ müssen für ausgelagerte Teile immer zwei Dateien geschrieben werden. Die **Headerdatei** und die eigentliche **Quelldatei**

## Headerdatei

Definiert **welche Funktionalität** der ausgelagerte Teil anbietet. z.B. Funktionen und Klassen

## Quelldatei

**Implementiert** die in der Headerdatei definierte Funktionalität

# Die Headerdatei

- Enthält einen sogenannten **Include Guard** der verhindert, dass die selbe Headerdatei öfter als ein mal inkludiert werden kann
- Enthält **Signaturen** aller Funktionen und Klassen
- Enthält Namen von **Konstanten** und **globalen Variablen**
- Enthält **#include** für anderen Headerdateien

## Include Guard

```
#ifndef EINDEUTIGER_NAME
#define EINDEUTIGER_NAME

// Alles andere kommt hier hin!

#endif
```

## Signaturen?

Signaturen teilen dem Compiler mit welche Funktionen und Klassen es gibt, ohne die Funktionalität bereits zu implementieren

# Was macht `#include` eigentlich?

- `#include` ersetzt die Zeile in der `#include` steht mit dem Inhalt der angegebenen Datei
- Der **Include Guard** verhindert, dass der Inhalt einer Datei nicht öfters verwendet wird.

## Beispiel

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

Die wiederholten Einfügungen von `iostream` werden vor dem compilieren wieder gelöscht.

# Signaturen von Funktionen

Bestehen einfach aus dem Teil einer Funktion vor den { }

## Beispiele

```
int add(int a, int b);  
bool is_even(int number);  
void print_my_stuff(MyClass &c);  
// ...
```

## Parameternamen

Die Signatur einer Funktion muss prinzipiell keine Parameternamen enthalten. Es ist für den Compiler nur wichtig, dass die Typen alle korrekt sind:

```
int add(int, int);  
bool is_even(int);  
void print_my_stuff(MyClass &);  
// ...
```

Der Übersichtlichkeit halber verwendet man sie generell aber trotzdem!

# Signaturen von Klassen

Die Signatur einer Klasse ist aufgebaut wie eine Klasse die wir bis jetzt gesehen haben, **enthält aber statt der Funktionen nur die Funktionssignaturen.**

## Beispiel für die ursprüngliche Player Klasse

```
class Player {
private:
    int _posx, _posy;
    string _img;
    int _fieldsx, _fieldsy;
    int _sizex, _sizey;

public:
    Player(int posx, int posy, string img, int fieldsx,
           int fieldsy, int sizex, int sizey);

    void move_left();
    void move_right();
    void move_up();
    void move_down();
    void draw();
};
```

- Inkludiert die zuvor beschriebene Headerdatei mit `#include "dateiname.h"` (**Achtung:** Die Anführungszeichen sind wichtig!)
- Implementiert die Funktionen und die Memberfunktionen von Klassen die wir in der Headerdatei beschrieben haben (siehe die folgenden beiden Beispiele)

# Auslagern von Funktionen — Beispiel

Wir wollen `is_prime` des folgenden Programms auslagern:

---

```
#include <iostream>
using namespace std;

bool is_prime(int number) {
    for (int i = 2; i < number / 2 + 1; i++) {
        if (number % i == 0)
            return false;
    }
    return true;
}

int main() {
    for (int i = 2; i < 100; i++) {
        if (is_prime(i)) {
            cout << i << " ist eine Primzahl" << endl;
        }
    }
}
```

# Auslagern von Funktionen — Beispiel

## prime.hpp

```
#ifndef PRIME_H
#define PRIME_H

bool is_prime(int number);

#endif /* PRIME_H */
```

## prime.cpp

```
#include "prime.hpp"

bool is_prime(int number) {
    for (int i = 2; i < number / 2 + 1; i++) {
        if (number % i == 0)
            return false;
    }
    return true;
}
```



# Auslagern von Funktionen — Beispiel

main.cpp

```
#include <iostream>
#include "prime.hpp"
using namespace std;

int main() {
    for (int i = 2; i < 100; i++) {
        if (is_prime(i)) {
            cout << i << " ist eine Primzahl" << endl;
        }
    }
}
```

# Auslagern von Klassen — Beispiel

Wir wollen **Average** des folgenden Programms auslagern:

```
#include <iostream>
using namespace std;

class Average {
private:
    double sum = 0.0;
    int count = 0;

public:
    void add(double val) {
        sum += val;
        count++;
    }

    double get_avg() { return sum / count; }
};

int main() {
    Average avg;
    avg.add(12.3);
    avg.add(11.7);
    avg.add(13.7);
    cout << avg.get_avg() << endl;
}
```

# Auslagern von Klassen — Beispiel

## average.hpp

```
#ifndef AVERAGE_H
#define AVERAGE_H

class Average {
private:
    double sum = 0.0;
    int count = 0;

public:
    void add(double val);
    double get_avg();
};

#endif /* AVERAGE_H */
```

# Auslagern von Klassen — Beispiel

## average.cpp

```
#include "average.hpp"

void Average::add(double val) {
    sum += val;
    count++;
}

double Average::get_avg() { return sum / count; }
```

## main.cpp

```
#include "average.hpp"
#include <iostream>
using namespace std;

int main() {
    Average avg;
    avg.add(12.3);
    avg.add(11.7);
    avg.add(13.7);
    cout << avg.get_avg() << endl;
}
```

# Globale Variablen

Globale Variablen müssen in der Headerdatei mit **extern** markiert und in der Quellcodedatei erzeugt werden

## Header

```
// ...  
extern int meine_globale_variable;  
// ...
```

## Quellcodedatei

```
#include "header.hpp"  
int meine_globale_variable = 42;
```

## Main

```
#include "header.cpp"  
#include <iostream>  
  
int main() {  
    std::cout << meine_globale_variable << std::endl;  
}
```

- Laden Sie sich die Datei `split_off.cpp` aus Sakai
- Lagern Sie alles bis auf die `main`-Funktion in eine Header- und Quellcodedatei aus