Einführung

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Einführung und Abgrenzung von C++`
Link auf GitHub:	https://github.com/TUBAF-Ifi-LiaScript/VL_SoftwareprojektRobotik/blob/master/00_Einfuehrung.md
Autoren	Sebastian Zug & Georg Jäger

Ausgangspunkt

Wie weit waren wir noch gekommen ... ein Rückblick auf die Veranstaltung Softwareentwicklung?

Ausgehend von der Einführung in C# haben wir uns mit:

- den Grundlagen der Objektorientierten Programmierung
- der Modellierung von konkreten Anwendungen
- der Koordination des Entwicklungsprozesses - Testen von Software, Versionsmanagement
- einer Einführung in die nebenläufige Programmierung

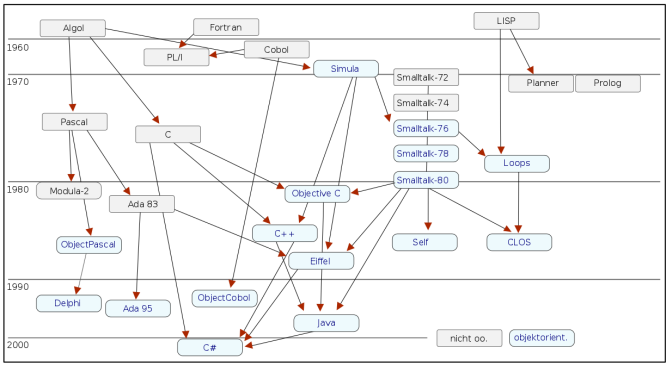
beschäftigt.

Warum sollten wir uns nun mit einer weiteren Programmiersprache beschäftigen? Welche Möglichkeiten eröffnen sich draus?

C++ ermöglicht sowohl die effiziente und maschinennahe Programmierung als auch eine Programmierung auf hohem Abstraktionsniveau. Der Standard definiert auch eine Standardbibliothek, zu der verschiedene Implementierungen existieren. Entsprechend findet C++ sowohl auf der Systemprogrammierungsebene, wie auch der Anwendungsentwicklung Anwendung.

C++ vs ...

Darstellung der Entwicklung von objektorientierten/nicht-objektorientieren Programmiersprachen [1]



Der Name C++ ist eine Wortschöpfung von Rick Mascitti, einem Mitarbeiter Stroustrups, und wurde zum ersten Mal im Dezember 1983 benutzt. Der Name kommt von der Verbindung der Vorgängersprache C und dem Inkrement-Operator „++“, der den Wert einer Variablen inkrementiert (um eins erhöht). Der Erfinder von C++, Bjarne Stroustrup , nannte C++ zunächst „C mit Klassen“ (C with classes).

Vortrag von Stroustrup auf der CppCon 2018: “Concepts: The Future of Generic Programming (the future is here)”

[1] Nepomuk Frädrich, Historie der objektorientierten Programmiersprachen, Wikimedia https://commons.wikimedia.org/wiki/File:History_of_object-oriented_programming_languages.svg

… C

C++ kombiniert die Effizienz von C mit den Abstraktionsmöglichkeiten der objektorientierten Programmierung. C++ Compiler können C Code überwiegend kompilieren, umgekehrt funktioniert das nicht.

Kriterium	C	C++
Programmierparadigma	Prozedural	Prozedural, objektorientiert, funktional
Kapselung	keine	Integration von Daten und Funktionen in `structs` und Klassen
Überladen	nein	Funktions- und Operator-Überladung
Programmierung	Präprozessor, C, Assemblercode	Präprozessor, C, C++, Assemblercode, Templates
Konzept von Zeigern	Pointer	(Smart-) Pointer, Referenzen
Integrationsfähigkeit	gering	hoch (namespaces)

Im folgenden soll die Verwendung eines `struct` unter C++ dem Bemühen um eine ähnliche Realisierung unter C mit dem nominell gleichen Schlüsselwort gegenübergestellt werden.

structExample.cpp

```cpp
#include <iostream>

struct Student{
  std::string name;
  int matrikel;
  void printCertificate(std::string topic);
};

void Student::printCertificate(std::string topic){
  std::cout << name << " passed " << topic;
}

int main()
{
  Student Humboldt {"Alexander Humboldt", 1798};
  Humboldt.printCertificate("Softwareentwicklung");
  return 0;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

structExample.c

```c
#include<stdio.h>
#include<string.h>

typedef struct student student;

struct student {
  char name[25];
  int matrikel;
  void (*print)(student *self, char *label);
};


void printCertificate(student * self, char* label){
  printf("%s passed %s", self->name, label);
}

int main()
{
  student Humboldt;
  strcpy(Humboldt.name, "Alexander von Humboldt");
  Humboldt.matrikel = 1798;
  Humboldt.print = &printCertificate;
  (Humboldt.print)(&Humboldt, "Softwareentwicklung");
  return 0;
}
```

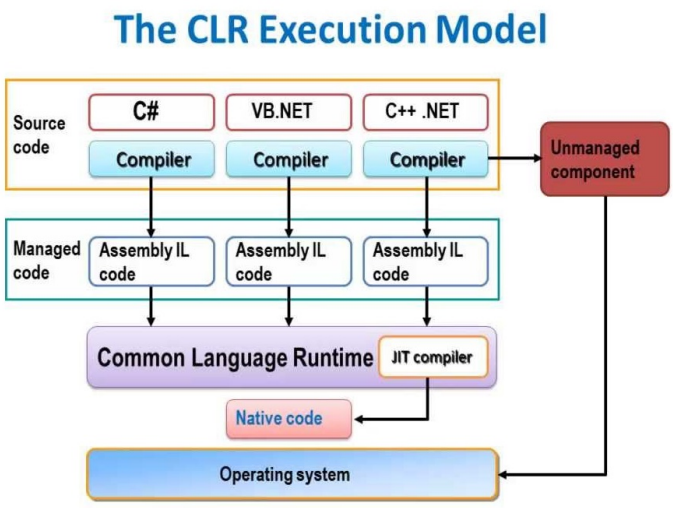
```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

"Encapsulation is pretty easy, polymorphism is doable - but inheritence is tricky" [Martin Beckett, www.stackoverflow.com]

... C#

Im Vergleich zwischen C++ und C# ergeben sich folgende Unterschiede / Gemeinsamkeiten

Zur Erinnerung sei noch mal auf das Ausführungskonzept von C# verwiesen.



.NET CLR Execution Model [1]

[1] Youtuber "MyPassionFor.NET", .NET CLR Execution Model, https://www.youtube.com/watch?v=gCHoBJf4htg

Am Beispiel

Unter anderem aus der überwachten Ausführung ergeben sich zentrale Unterschiede beim Vergleich von C# und C++:

OutOfRange.cs

```cs
using System;

public class Program
{
  public static void printArray(int[] array){
  for (int i = 0; i <= array.Length; i++)
    Console.WriteLine(array[i]);
  }
  public static void Main(string[] args)
  {
    int[] array = {1, 2, 3, 4, 5};
    printArray(array);
  }
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

OutOfRange.cpp

```cpp
#include <iostream>

void printArray(int array []){
  for (unsigned int i = 0; i < 51; i++){
    std::cout << array[i] << ",";
  }
}

int main()
{
  int array [] {1,2,3,4,5};
  printArray(array);
  return 0;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

Aspekt	**C++**	**C#**
Entwicklung	ab 1979 von Bjarne Stroustrup, Standardisierung 1998, aktueller Stand C++17 (von vielen Compilern noch nicht unterstützt)	ab 2001 von Microsoft entwickelt, ab 2003 ISO genormt
Kompilierung	Programmcode wird auf spezifischen Maschinencode abgebildet	C# Compiler generiert übergreifende Coderepräsentation Common Intermediate Language (CLI)
Ausführungsumgebung	Unmittelbar auf Prozessorebene	in Laufzeitumgebung Common Language Runtime (CLR), die einen Just-in-Time-Compiler umfasst der die Übersetzung in Maschinencode übernimmt.
Plattformen	Compiler für jedwede Architektur und Betriebssysteme	setzt .NET Ausführungsumgebung voraus
Speicher Management	Kein Speichermanagement	die CLR umfasst unter anderem einen Garbage Collector
Verwendung von Pointern	Elementarer Bestandteil des Programmierkonzepts	nur im `unsafe` mode
Objektorientierung	Fokus auf objektorientierte Anwendungen	pur objektorientiert (zum Beispiel keine globalen Funktionen)
Vererbung		alle Objekte erben von einer Basisklasse `object`
	unterstützt Mehrfachvererbung (ersetzt Interfaces)	keine Mehrfachvererbung
Standard Zugriffsattribut	`public` für structs, `private` für Klassen	`private`

Elemente der Sprache C++

An dieser Stelle wir keine klassische Einführung in C/C++ erfolgen, vielmehr sei dabei auf die einschlägigen Tutorials und Literaturbeispiele verwiesen, die in Eigenregie zu erarbeiten sind. An dieser Stellen werden die Basiskonzept in einem groben Überblick und anhand von Beispielen eingeführt.

C++11 Schlüsselworte

Die Sprache C++ verwendet nur etwa 60 Schlüsselwörter („Sprachkern“), manche werden in verschiedenen Kontexten (static, default) mehrfach verwendet.

Bedeutung	Inhalt	Schlüsselwort
Grunddatentypen	Wahrheitswerte	bool, true, false
	Zeichen und Zahlen	char, char16_t, char32_t, wchart_t
	Zahlen	int, double, float
	weitere	auto, enum , void
Modifizierer	Platzbedarf	long, short
	Vorzeichen	signed, unsigned
	Manipulierbarkeit	const, constexpr, mutable, volatile
Zusammengesetzte Typen	Klassen, Strukturen	class, struct, union, explicit, this, virtual
	Zugriffsrechte	friend, private, protected, public
Typinformationen		alignof, decltype, sizeof, typeid, typename
		const_cast, dynamic_cast, reinterpret_cast, static_cast
Ablaufsteuerung	Schleifen	do, for, while
	Verzweigungen	if, else, default, switch, case
	Sprünge	break, continue, goto
	Ausnahmebehandlungen	catch, noexcept, static_assert, throw, try
Assemblercode		asm
Speicherhandling		delete, new, nullptr
Funktionen		inline, operator, return
Namensbereiche und Alias		namespace, using, typedef
Schablonen		template

In den folgenden Lehrveranstaltungen sollen einzelne Aspekte dieser Schlüsselworte anhand von Beispielen eingeführt werden.

Variablenverwendung

Datentypen

Kategorie	Bezeichner	Bemerkung
Ganzzahl	`int`, `short`, `long`, `long long`	jeweils als `signed` und `unsigned`
Fließkomma	`double`, `float`, `long double`	
Wahrheitswerte	`bool`	
Zeichentypen	`char`, `char16_t`, `char32_t`	Die Größe von `char` entspricht dem kleinsten Ganzzahldatentyp
Referenzen	Indirektion mit `&`	
Zeiger	Indirektion mit `*`	

Während C# spezifische Größenangaben für die Variablen trifft Link sind die maximalen Werte für C++ Programme systemabhängig.

Auf die realisierten Größen kann mit zwei Klassen der Standardbibliothek zurückgegriffen werden.

1. `climits.h` definiert ein Set von Makrokonstanten, die die zugehörigen Werte umfassen. Unter C++ wird diese Bibliothek mit `climits.h` eingebettet, da `limits` durch einen eignen Namespace besetzt ist Link mit Übersicht
2. `numeric_limits.h` spezifiziert Templates für die Bereitstellung der entsprechenden Grenzwerte und ist damit deutlich flexibler.

Hello.cpp

```cpp
// numeric_limits example
#include <iostream>     // std::cout
#include <limits>       // std::numeric_limits
#include <climits>

int main () {
  std::cout << "Bits for char: " << CHAR_BIT << '\n';
  std::cout << "Minimum value for char: " << CHAR_MIN << '\n';
  std::cout << "Maximum value for char: " << CHAR_MAX << '\n';
  std::cout << "-----------------------------------------------------\n";
  std::cout << "Minimum value for int16_t: " << INT16_MIN << '\n';
  std::cout << "Maximum value for int16_t: " << INT16_MAX << '\n';
  std::cout << "-----------------------------------------------------\n";
  std::cout << std::boolalpha;
  std::cout << "Minimum value for int: " << std::numeric_limits<int>::min()
    << '\n';
  std::cout << "Maximum value for int: " << std::numeric_limits<int>::max()
    << '\n';
  std::cout << "int is signed: " << std::numeric_limits<int>::is_signed <<
    '\n';
  std::cout << "Non-sign bits in int: " << std::numeric_limits<int>::digits
    << '\n';
  std::cout << "int has infinity: " << std::numeric_limits<int
    >::has_infinity << '\n';
  return 0;
}
```

```
Bits for char: 8
Minimum value for char: -128
Maximum value for char: 127
-----------------------------------------------------
Minimum value for int16_t: -32768
Maximum value for int16_t: 32767
-----------------------------------------------------
Minimum value for int: -2147483648
Maximum value for int: 2147483647
int is signed: true
Non-sign bits in int: 31
int has infinity: false
```

Eine spezifische Definition erfolgt anhand der Typen:

Typ	Beispiel	Bedeutung
`intN_t`	`int8_t`	"... denotes a signed integer type with a width of exactly N bits." (7.18.1.1.)
`int_leastN_t`	`int_least32_t`	"... denotes a signed integer type with a width of at least N bits." (7.18.1.2.)
`int_fastN_t`	`int_fast32_t`	"... designates the fastest unsigned integer type with a width of at least N." (7.18.1.3.)

Hello.cpp

```cpp
#include <iostream>
#include <typeinfo>

int main()
{
  std::cout<<typeid(int).name() << " - "<<  sizeof(int) << " Byte \n";
  std::cout<<typeid(int32_t).name()  << " - "<<  sizeof(int32_t) << " Byte
    \n";
  std::cout<<typeid(int_least32_t).name() << " - "<<  sizeof(int_least32_t)
    << " Byte \n";
  std::cout<<typeid(int_fast32_t).name() << " - "<<  sizeof(int_fast32_t)
    << " Byte \n";
}
```

```
i - 4 Byte
i - 4 Byte
i - 4 Byte
l - 8 Byte
```

Dazu kommen entsprechende Pointer und max/min Makros.

Die Herausforderung der architekturspezfischen Implmementierungen lässt sich sehr schön bei der Darstellung von Größenangaben von Speicherinhalten, wie zum Beispiel für Arrays, Standardcontainer oder Speicherbereiche illustrieren. Nehmen wir an Sie wollen eine Funktion spezifizieren mit der Sie einen Speicherblock reservieren. Welche Einschränkungen sehen Sie bei folgendem Ansatz:

memcpy.cpp

```cpp
void memcpy(void *s1, void const *s2, int  n);
```

`size_t` umgeht dieses Problem, in dem ein plattformabhäniger Typendefinition erfolgt, die maximale Größe des adressierbaren Bereiches berücksichtigt.

Deklaration, Definition und Initialisierung

In der aufgeregten Diskussion werden die folgenden Punkte häufig vermengt, daher noch mal eine Wiederholung:

- Deklaration ... Spezifikation einer Variablen im Hinblick auf Typ und Namen gegenüber dem Compiler
- Definition ... Anlegen von Speicher für die Variable
- Initialisierung ... Zuweisung eines Anfangswertes

Anweisung	Wirkung
`int i`	Definition und Deklaration der Variablen i
`extern int i`	Deklaration einer Variablen i
`int i; i = 1;`	Initialisierung nach Deklaration
`const int number = 1`	Deklaration und Initialisierung einer konstanten Variablen

Bei unveränderlichen Werten muss die Initialisierung immer unmittelbar mit der Definition und Deklaration erfolgen.

In C++11 wurde die Initialisierung, die sich bisher für verschiedenen Kontexte unterschied, vereinheitlicht. Sie kennen die Angabe von Initialisierungswerten mittels `{}` bereits von arrays unter C.

```cpp
int numbers[] = { 1, 2, 4, 5, 9 };
```

Anweisung	Bedeutung
`int i{};`	uninitialisierter Standardtyp
`int j{10};`	initialisierter Standardtyp
`int a[]{1, 2, 3, 4}`	aggregierte Initialisierung
`X x1{}; X x2();`	Standardkonstruktor eine individuellen Typs
`X x3{1,2}; X x4(1,2);`	Parameterisierter Konstruktor
`X x5{x3}; X x6(x3);`	Copy-Konstruktor

Die `auto`-Schlüsselwort weist den Compiler an den Initialisierungsausdruck einer deklarierten Variable oder einen Lambdaausdrucksparameter zu verwenden, um den Typ herzuleiten. Damit ist eine explizite Angabe des Typs einer Variablen nicht nötig. Damit steigert sich die Stabilität, Benutzerfreundlichkeit und Effizienz des Codes.

auto.cpp

```cpp
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>
#include <iostream>
#include <string>
#include <memory>
#include <cstdlib>

std::string demangle(const char* mangled)
{
    int status;
    std::unique_ptr<char[], void (*)(void*)> result(
      abi::__cxa_demangle(mangled, 0, 0, &status), std::free);
    return result.get() ? std::string(result.get()) : "error occurred";
}

template<class T>
void foo(T t) { std::cout << demangle(typeid(t).name()) << std::endl; }

int main()
{
  auto var1 = 4;
  auto var2 {3.14159};
  auto var3 = "Hallo";
  auto var4 = new double[10];

  // Datentyp der Variablen ausgeben
  std::cout << typeid(var3).name() << std::endl;

  // ein bisschen hübscher .... für Linux Systeme
  foo(var1);
  foo(var2);
  foo(var3);
  foo(var4);

  // aufräumen nicht vergessen
  delete[] var4;
}
```

```
PKc
int
double
char const*
double*
```

Wir werden `auto` im folgenden auch im Zusammenhang mit Funktionsrückgaben und der effizienten Implementierung von Schleifen einsetzen.

Zeichenketten, Strings und Streams

Verwendung von Zeichenketten und Strings

Aus historischen Gründen kennt C++ zwei Möglichkeiten Zeichenketten darzustellen:

- `const char *` aus dem C-Kontext und
- `std::string` die Darstellung der Standardbibliothek

Für die Ausgabe einer nicht veränderlichen Zeichenkette kann man nach wie vor mit dem `const char` Konstrukt arbeiten:

cStyleStrings.cpp

```
#include <iostream>

int main()
{
    //             01234567890123456789012
    char text[] = "Softwareprojekt Robotik";
    std::cout << sizeof(text) << std::endl;
    std::cout << "Erstes Zeichen:  " <<  text[0] << "\n";
    std::cout << "Sechstes Zeichen:" << *(text+5) << "\n";
    std::cout << "Letztes Zeichen: " << text[sizeof(text)-2] << "\n";
    std::cout << text << "\n";
}
```

```
24
Erstes Zeichen:  S
Sechstes Zeichen:a
Letztes Zeichen: k
Softwareprojekt Robotik
```

Der Hauptunterschied in der Anwendung der `std::string`-Klasse besteht darin, dass Sie den String nicht über einen char-Zeiger manipulieren können. Ein Pointer auf ein Objekt der Klasse `string` zeigt ja nicht auf den ersten Buchstaben, sondern auf das Objekt, das die Zeichen verwaltet. Darüber steht eine Zahl von Elementfunktionen wie `length()`, `insert(n,s)`, `find(s)` zur Verfügung.

CplusplusStrings.cpp

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

int main()
{
    std::string Alexander {"Alexander von Humboldt"};
    std::string uniName {"TU Bergakademie"};
    std::cout << Alexander + ", " + uniName << "\n";
    return 0;
}
```

```
Alexander von Humboldt, TU Bergakademie
```

Wie beispielhaft gezeigt können `char` Arrays und `string` Objekte miteinander in einfachen Operatoren verglichen werden. Für das Durchlaufen der Zeichenreihe steht ein eigenes Zeigerkonstrukt, der Iterator bereit, der durch `.begin()` und `.end()` in seiner Weite definiert wird. In Kombination mit `auto` können sehr kompakte Darstellungen umgesetzt werden.

> MERKE: Für konstante Textausgaben kann gern der `const char *` Typ verwendet darüber hinaus kommen Instanzen der `std::string` Klasse zum Einsatz.

Diskussion cout vs. printf

printfVscout.cpp

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

struct Student{
    std::string Vorname;
    std::string Name;
    int Matrikel;
};

std::ostream &operator << (std::ostream &out, const  Student &m) {
    out << m.Name << " " << m.Matrikel;
    return out;
}

int main()
{
    Student Alexander {"Alexander", "von Humboldt",  1791};
    Student Bernhard {"Bernhard", "von Cotta", 1827};
    printf("%-20s %-20s %5s\n" , "Name" , "Vorname" , "Id" );
    printf("%-20s %-20s %5d\n" , Alexander.Name.c_str(),
                                 Alexander.Vorname.c_str(),
                                 Alexander.Matrikel);
    printf("%-20s %-20s %5d\n" , Bernhard.Name.c_str(),
                                 Bernhard.Vorname.c_str(),
                                 Bernhard.Matrikel);


    std::cout << Alexander;
    return 0;
}
```

```
Name                 Vorname                Id
von Humboldt         Alexander            1791
von Cotta            Bernhard             1827
von Humboldt 1791
```

"While cout is the proper C++ way, I believe that some people and companies (including Google) continue to use printf in C++ code because it is much easier to do formatted output with printf than with cout. " StackOverflow-Beitrag

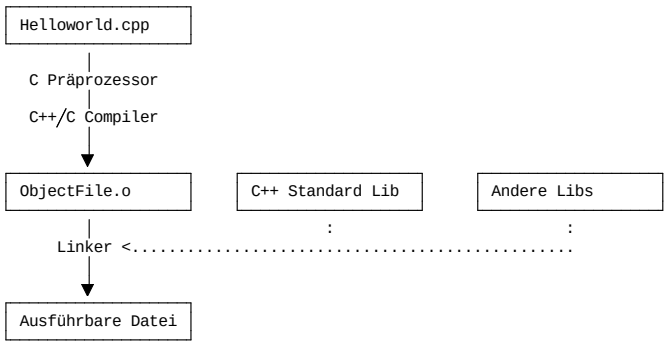
Hinsichtlich der Performance existieren einige sehr schöne, wenn auch etwas ältere Untersuchungen, die zum Beispiel im Blog von Filip Janiszewski beschrieben werden Link. Die Ergebnisse decken sich mit den vorangegangenen Empfehlungen.

Wiederholung: Was passiert mit "Hello World"?

reducedHello.cpp

```cpp
//#include <iostream>

int main()
{
    //std::cout << "Hello, World!";
    return 1;
}
```



Durchlaufen Sie zunächst die Toolchain mit dem `reducedHello.cpp` Beispiel. Erklären Sie die die Inhalte der Einträge in

```
g++ reducedHello.cpp -o Hello       // Realisiert die gesamte Kette in einem
  Durchlauf
ldd Hello                           // Liste der referenzierten Bibliotheken
g++ -E reducedHello.cpp -o Hello.ii // Stellt die Präprozessorausgabe bereit
wc -l reducedHello.ii               // Zeilenzahl der Präcompilierten Datei
g++ -S reducedHello.cpp -o Hello.S  // Stellt den Assemblercode bereit
g++ -c reducedHello.cpp -o Hello.o  // Generiert das Objektfile für HelloWorld
  .cpp
```

Präprozessor

> Ein Wort der Warnung ...

Der Präprozessor durchläuft alle Quelltextdateien (*.cpp) noch vor der eigentlichen Kompilierung und reagiert auf enthaltene Präprozessordirektiven: Zeilen die mit "#" beginnen.

Pro Zeile kann lediglich eine Direktive angegeben werden. Eine Direktive kann sich jedoch über mehrere Zeilen erstrecken, wenn der Zeilenwechsel mit einem "\" versehen wird.

Typische Anwendungen:

- #include - Inkludieren weiterer Header-Dateien
- #define - Definition von Macros
- #if-#else-#endif - Bedingte Kompilierung

Die Nutzung von Macros in C++ sollte man limitieren:

- Bugs sind schwer zu finden - Quelltext wird ersetzt bevor er kompiliert wird
- Es gibt keine Namespaces für Macros - Bei unglücklicher Namensgebung können Funktionen/Variablen/Klassen aus C++ durch Präprozessormacros ersetzt werden.
- Nichtintuitive Nebeneffekte:

Hello.cpp

```cpp
#include <iostream>
#define SQUARE(x) ((x) * (x))

int main()
{
    int x = 5;
    std::cout << "SQUARE(x) = " << SQUARE(x++) << std::endl;
    return 0;
}
```

```
SQUARE(x) = 30
```

Aufgabe bis zur nächsten Woche

- Installieren Sie eine C++ Entwicklungsumgebung auf dem Rechner (Windows mit wsl2 oder cygwin)
- Erweitern Sie die Untersuchung auf ein Projekt mit mehrere Dateien. Wie müssen Sie vorgehen um hier eine Kompilierung zu realiseren? Wie kann sie Dabei ein Makefile unterstützen?
- Arbeiten Sie sich in die Grundlagen der C++ Programmierung (Ausgaben, Eingaben, Programmfluss, Datentypen) ein.

Schreiben Sie ein Programm, dass eine positive ganze Zahl als Parameter entgegen nimmt und die größte Primzahl bestimmt, die kleiner als diese Zahl ist.