C++

Warum?

- Deutlich schnellere Ausführung als bei Python
- Oft für Echtzeit-Anwendungen geeignet (Steuerung von Messtechnik, Auto, Flugzeug)
- Möglichkeiten, Erwartungen und Versprechen über die Eigenschaften von Funktionen und Variablen zu formulieren
- Zu C++ gehört auch die Standard-Bibliothek (Standard Library)
  - Container (Listen, Arrays, ...)
  - Algorithmen (Akkumulieren, Suchen, Sortieren, ...)
  - Ein/-Ausgabe
  - ...

C++-Code muss übersetzt werden

- Es gibt keinen (verbreiteten) Interpreter.
- Der Quellcode wird vor der Ausführung kompiliert
- Es gibt verschiedene Compiler
  - GNU Compiler Collection
  - LLVM/Clang
  - Microsoft Visual C++ Compiler
- *programm.cpp* kompilieren: `bash  c++ -o programm programm.cpp`
- Dabei kommt eine Binärdatei namens *programm* heraus, die man aufrufen kann `bash  ./programm`
- Weitere hilfreiche Parameter
  - `-std=c++17` schaltet den C++17-Standard an (den wir benutzen)
  - `-O2` schaltet einige Optimierungen an
  - `-Wall` schaltet mehr sog. *Warnungen* ein. Oft für Sachen, die aus historischen Gründen nicht verboten sind, aber eine wirklich schlechte Idee sind.
  - `-Werror` macht aus *Warnungen Fehler*.

Einfacher Primzahlfinder in C++

```
#include <iostream>

int main(int argc, char **argv) {
  using std::cin;
  using std::cout;

  cout << "Upper limit: ";
```

```
  int limit;
  cin >> limit;

  for (int candidate = 2; candidate < limit + 1; candidate++) {
    bool is_prime = true;
    for (int test = 2; test < std::min(candidate, limit / 2 + 1); test++) {
      if (candidate % test == 0) {
        is_prime = false;
        break;
      }
    }
    if (is_prime) {
      std::cout << candidate << " is a prime" << std::endl;
    } else {
      std::cout << candidate << " is not a prime" << std::endl;
    }
  }
}
```

Einige wichtige Unterschiede zu Python

- Statische Typisierung
  - Der Datentyp von Variablen wird zur Compile-Zeit entschieden
  - Bei (oder vor) der ersten Verwendung muss der Datentyp einer Variable deklariert werden
  - Der Datentyp einer Variable kann sich nicht zur Laufzeit ändern
- Strichpunkt nach jeder Anweisung
- Code-Blöcke werden durch Klammern `{}` markiert
- Bedingungen bei `if` und `while` stehen in runden Klammern
- Literale `true` und `false` werden kleingeschrieben

Programmstruktur

- Includes: Einbinden von sog. Headern, in denen angegeben ist, welche Funktionen eine Bibliothek zur Verfügung stellt `c++ #include <string> // String-Funktionen aus der Standard-Bibliothek`
- Globale Variablen: Deklaration von Variablen, die überall im Programm sichtbar und schreibbar sind. So selten wie möglich benutzen! (Warum?)
- Definitionen von Klassen und Funktionen
- Der Quellcode wird von oben nach unten gelesen.
- Dinge, die man nutzt, müssen vorher deklariert oder inkludiert worden sein.

- Die Reihenfolge der Deklarationen und Includes muss dem Rechnung tragen
- Die `main()`-Funktion wird ausgeführt, wenn das Programm startet.
  - Deklaration:

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

  - Die Argumente der Funktion sind Anzahl und Inhalt der Kommandozeilenargumente
  - Der Rückgabewert ist der Exit-Code, der in der Shell ankommt.

Datentypen

- Ganzzahlen: `int`, `long`, `unsigned int`, ...
- Fließkomma-Zahlen
  - *`double`*: doppelte Genauigkeit (fast immer die richtige Wahl)
  - `float`: einfache Genauigkeit (anders als bei Python!)
- Zeichenketten
  - `std::string`. Beste Wahl. (`<string>` muss inkludiert werden)
  - `char[n]`: ein ungeschütztes Array von Zeichen. Das NULL-Zeichen markiert das Ende der Zeichenkette
  - `const char[n]`: Der Datentyp eines String-Literals (`"hallo"`)
- Bool'sche Variable (true/false): `bool`

Variablen

- Müssen bei der ersten Verwendung mit Datentyp deklariert werden `C++ #include <string> int a = -5; double b = 1.0; double c = b; // Copy by value! double d; // d ist ab jetzt nutzbar, der Wert ist aber undefiniert! std::string h = "hallo";`
- Der Datentyp kann sich nach der Deklaration nicht mehr ändern
- Wird eine Variable `const` deklariert, ist auch ihr Wert konstant `C++ #include <cmath> // ... const double sqrt_two = std::sqrt(2.0);`
  - Immer `const` nutzen, wenn es möglich ist. Der Compiler meldet sich, wenn man den Wert aus Versehen nachträglich ändert.
- Referenzen `c++ int a = 1; int & b = a; // Erzeugt eine Referenz auf a, Copy by reference`
- Auf der rechten Seite muss eine Variable o.Ä stheen, kein Literal
- ändert man `b`, ändert sich auch `a` und umgekehrt;
- Mit einer `const`-Referenz lässt sich eine schreibgeschützte Kopie erstellen `C++ int a = 0; const int & b = a;`
- Automatische Ableitung des Datentyps

  - bei als `auto` deklarierte Variablen wird der Typ durch die rechte Seite der Zuweisung bestimmt
    ```
    auto a = 5*3; // a ist dann vom Typ int
    ```
  - Nur nutzen, wo es der Verständlichkeit nicht schadet.

Berechnungen, Operatoren

- Grundrechenarten: `+ - * /`
  - Der Rückgabewert ist der "beste" gemeinsame Typ:
    - Für die Operanden vom Typ `int` und `int` ist das Ergebnis `int`
    - Für `int`, `double` ist das Ergebnis `double`
    - Bei der Integerdivision `int / int` wird abgeschnitten
- Divisionsrest: `%`
- Vergleich:
  - `== !=`: gleich, ungleich
  - `<`, `>`, `<=`, `>=`
- Zuweisung: `=`
- Stream-Operatoren:
  - `<<` und `>>` schaufeln Daten in Richtung der Pfeile
  - für Datentypen, die das Unterstützen
  - z.B. die Ein-/Ausgabe-Streams `std::cin` und `std::cout`

Bedingungen

```
if (bedingung) {
  ...
} else {
  ...
}
```

- Die Bedingung steht in Klammern
- Vergleichsoperator ist `==`
- ACHTUNG: `c++ if (a = b) { }` ist in C++ legal, entspricht aber `c++ a=b; if (a) { }` Es ist also eine Zuweisung `a = b`, kein Vergleich `a == b`.

For-Schleifen

- Die klassische Version hat drei Anweisungen im Schleifenkopf
  - Initialisierung. Bsp.: `int i = 0`
  - Bedingung. Bsp.: `i<10`
  - Aktion. Bsp.: `i++` oder `i = i + 1` oder `i += 1`

```
for (int i = 0; i < 10; i++) {
}
```

- Schleife über Elemente eines Containers (wie in Python, Range-based loop)
  – machen wir, sobald wir Container kennen

```
// Sieht oft folgendermaßen aus
for (auto x : container) {
}
```

While-Schleife

```
while (bedingung) {
}
```

Beispiel:

```
int i = 0;
while (i < 10) {
  i++;
}
```

Schleifen: break und continue

- `break` steigt aus der Schleife aus
- `continue` springt direkt zum nächsten Durchlauf
- wie in Python und Bash

Funktionen

- Ermöglichen, Code für abgegrenzte Aufgaben mehrmals zu verwenden
- Müssen einen Datentyp für den Rückgabewert angeben (void, wenn nichts zurückgegeben werden soll)
- Können Argumente haben. Typ und Name müssen angegeben werden: `int a, double b`
- Die letzten Argumente können einen Standardwert haben `int a, double b = 0`
- Die Rückgabe des Ergebnis erfolgt mit `return`. Kann bei Return-Type `void` ausgelassen werden

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

double addieren(double a, double b) {
  return a + b;
}
```

```
void ausgabe(int x) {
   std::cout << "x ist " << x << std::endl;
}

int main(int arc, char** argv) {
  double vier = quadrat(2);
  auto summe = addieren(1, 3); // Welchen Typ hat die Variable summe?
  return 0;
}
```

Templates: Vorlagen für Funktionen

- Vorlagen für Funktionen (und Klassen), die für mehrere Datentypen nutzbar sind
- Ein Funktions-Template ist nur der Bauplan
- Nur die Varianten, die aufgerufen werden, werden auch compiliert

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

template <typename T>
T addieren(T a, T b) {
  return a + b;
}

int main(int argc, char** argv) {
  std::cout << "1 + 2 = " << addieren(1,2) << std::endl; // T ist int
  std::cout << "1.0 + 2.1 = " << addieren(1.0,2.1) << std::endl; // T ist double
  std::string a = "Hallo ", b = "Welt";
  std::cout << addieren(a, b) << std::endl; // T ist std::string
  return 0;
}
```

Funktions-Templates II

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

template <typename T>
T addieren(T a, T b) {
  return a + b;
}
```

Welche Beschränkungen gibt es bei der Nutzung dieses Funktions-Templates?

Templates:: Fehlermeldungen I

```
template <typename T>
T addieren(T a, T b) {
  return a + b;
}
```

- Zwei ungleiche Typen `addieren(1, true)` `/home/weeber/x.cpp: In function ‘int main(int, char**)’: /home/weeber/x.cpp:10:19: error: no matching function for call to ‘addieren(int, bool)’ addieren(1, true); ^ /home/weeber/x.cpp:5:3: note: candidate: template<class T> T addieren(T, T) T addieren(T a, T b) { ^~~~~~~~ /home/weeber/x.cpp:5:3: note: template argument deduction/substitution failed: /home/weeber/x.cpp:10:19: note: deduced conflicting types for parameter ‘T’ (‘int’ and ‘bool’) addieren(1, true); ^`

Templates:: Fehlermeldungen II

```
template <typename T>
T addieren(T a, T b) {
  return a + b;
}
```

- Nutzung mit Typen, die den +-Operator nicht unterstützen
  - `addieren("a", "b")`
  - String-Literale haben den Typ `const char*`, wenn man sie nicht auf einen `std::string` zuweist.
  - Für `const char[n]` ist Addition nicht definiert

```
/home/weeber/x.cpp: In instantiation of ‘T addieren(T, T) [with T = const char*]’:
/home/weeber/x.cpp:10:20:   required from here
/home/weeber/x.cpp:6:12: error: invalid operands of types ‘const char*’ and ‘const
   return a + b;
          ~~^~~
```

Standard-Bibliothek: Container

- Speichern mehrere Objekte
- Haben unterschiedliche Garantien bezüglich der Laufzeit von Operationen
- Sind Template-Klassen, können also für fast alle Datentypen verwendet werden

- Speichern i.d.R. lauter Elemente des gleichen Typs (Ausnahme: Vererbung, später bei Objektorientierung)
- `std::array`: Liste feste Länge, durchnummerierte Elemente
- `std::vector`: Liste variable Länge, durchnummerierte Elemente, wie Python-`list`
- `std::unordered_set`: (deutsch: Menge) Ein Wert ist nicht oder genau einmal drin, wie Python-`set`
- `std::unorderd_map`: (deutsch: Abbildung) Schlüssel-Wert-Paare, wie Python-`dict`
- Es gibt mehr. Siehe https://cppreference.com/w/cpp/container

`std::vector`

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

int main() {
  std::vector<double> v_d = {1.0, 0.5, -1};
  v_d.push_back(42); // Element anhängen
  std::cout << "2. Element: "<< v_d[1] << std::endl;

  // Iterieren
  for (auto d: v_d) {  // Range-based Loop, wie Python
     std::cout << "v_d enthält " << d << std::endl;
  }

  // Leeren
  v_d.clear();
  std::cout << "Länge des Vektors ist " << v_d.size() << std::endl;
}
```