Teil 5
fortgeschrittene Sprachelemente

- Arrays
- Container
- Typdefinitionen
- Schleifen mit Iteratoren
- Sprungbefehle
- Switch / Case
- Aufzählungen
- Typumwandlungen
- Referenzen

Arrays

C++ 11

```
#include <array>

std::array<int, 5> primes = {3, 2, 11, 7, 5};
cout << primes[1] << endl;
```

- C++ anstatt C
- kennt seine eigene Länge
- unterstützt Iteratoren und damit `std::sort` etc.
- ermöglicht Move-Semantics

```
#include <algorithm>

std::sort(primes.begin(), primes.end());
cout << primes.size() << endl;
```

Container

```
#include <vector>

std::vector<int> primes = {2, 3, 5, 7, 11};
cout << primes[1] << endl;
primes.push_back(13);
primes.clear();
primes.resize(100);
```

C++ 11

- dynamische Länge
- schneller Indexzugriff, aber Einfügen / Löschen in der Mitte langsam (O(n))

```
#include <list>

std::list<int> primes = {2, 3, 5, 7, 11};
```

C++ 11

- Zugriff auf Elemente via Index langsam (O(n))

Container

```
#include <map>

std::map<int, string> primes = {{2, "two"}, {3, "three"}};
primes[5] = "five";
primes.erase(2);
```

C++ 11

- Keys müssen sortierbar sein (Vergleichsoperatoren)
- Zugriff sehr schnell (O(log(n))) durch Red-Black-Tree

```
#include <unordered_map>

std::unordered_map<int, string> primes;
```

C++ 11

- Keys brauchen Hash-Funktion
- Zugriff extrem schnell (O(1)), aber mehr Platzbedarf

Aufgaben

Aufgabe 13A

- eine Funktion soll eine Liste mit irgendwelchen Zahlen zurückgeliefen
- optional: der Aufrufer bestimmt eine „Zahlenreihe“, z.B. Vielfache von 3
- Das Hauptprogramm soll den Inhalt dieses Containers ausgeben

Aufgabe 13B

- eine Funktion soll die Liste von Primzahlen berechnen und in einem geeigneten Container zurückliefern
- Das Hauptprogramm soll den Inhalt dieses Containers ausgeben

Typdefinitionen

```
typedef std::list<std::string> StringList;
StringList* strings = new StringList;

typedef std::shared_ptr<StringList> StringListPtr;
StringListPtr strings(new StringList);
```

- Vereinfachung insbesondere komplexerer Typen
- bei häufiger Wiederverwendung

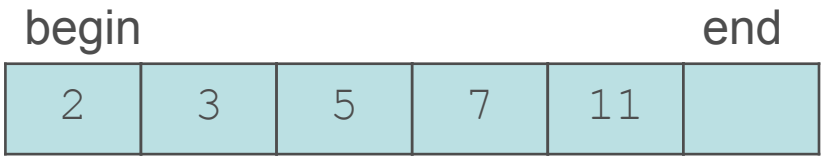
Iterator-Schleifen

C++ 11

```
std::vector<int> primes = {2, 3, 5, 7, 11};
for (int n : primes) { cout << n; }
```

- „range based for loops“ funktioniert mit allen Containern, die `begin()` und `end()` mitbringen
- lässt sich mit `auto` nutzen

```
std::vector<int> primes = {2, 3, 5, 7, 11};
for (std::vector<int>::iterator it = primes.begin();
  it != primes.end(); ++it) { cout << *it; }
```



Aufgaben

```
std::vector<int> primes = {2, 3, 5, 7, 11};
for (int n : primes) { cout << n; }
```

Aufgabe 14A

- Lassen sich im vorigen Programm „range based for loops“ nutzen?

Aufgabe 14B

- Lassen sich im vorigen Programm „range based for loops“ nutzen?

Sprungbefehle

```
string text = "Anna & Boris, Claudia & Dirk";
for (char c : text)
{
  if (c == ' ') continue;
  if (c == ',') break;
  cout << c;
}
```

- `break` beendet die Schleife sofort
- `continue` überspringt den Rest der Schleife, um mit der nächsten Iteration fortzufahren
- beziehen sich auf das innere `for` / `while` / `do/while`
- `if` / `else` und andere Codeblöcke spielen keine Rolle

Sprungbefehle

```
string lines[] = {"Anna & Boris", "Claudia & Dirk", "<Ende>"};
for (auto line : lines)
{
  for (char c : line)
  {
    if (c == '<') goto end;
    cout << c;
  }
}
end:
```

- `goto` wird seltenst benutzt
- Ausnahme vielleicht der Sprung aus verschachtelten Schleifen
- bessere Lösung z.B. separate Funktion und `return`

Fallunterscheidung

```
void print(int number, string singular, string plural);
{
  switch (number)
  {
    case 0:
      cout << "keine " << plural;
      break;
    case 1:
      cout << "1 " << singular;
      break;
    default:
      cout << number << " " << plural;
      break;
  }
}

print(n, "Elefant", "Elefanten");
```

- `break` darf nicht vergessen werden !!!

Fallunterscheidung

```
string quantify(int number, string singular, string plural)
{
  switch (number)
  {
    case 0:  return string("keine ") + plural;
    case 1:  return string("1 ") + singular;
    default: return std::to_string(number) + " " + plural;
  }
}

cout << quantify(n, "Elefant", "Elefanten") << endl;
```

- `return` in `switch/case` ist viel klarer

Aufzählungen

```
enum DayOfWeek {monday, tueday, wednesday, thursday, friday,
saturday, sunday};

bool isWeekend(DayOfWeek day)
{
  switch (day)
  {
    case saturday:
    case sunday:    return true;
    default:        return false;
  }
}

DayOfWeek day = friday;
++day;
```

Semikolon

kein Namespace

nicht möglich

```
enum Direction {north = 0, east = 90, south = 180, west = 270};
Direction home = south;
```

Aufzählungen

C++ 11

```
enum class DayOfWeek {monday, tueday, wednesday, thursday, friday,
saturday, sunday};

bool isWeekend(DayOfWeek day)
{
  switch (day)
  {
    case DayOfWeek::saturday:
    case DayOfWeek::sunday:    return true;
    default:                   return false;
  }
}

DayOfWeek day = DayOfWeek::friday;
int nr = day;
```

Namespace

nicht möglich

Typumwandlungen

```
// C-Style
int x = 3, y = 2;
double r = (double)x / (double)y;
```

```
// C-Style
Direction home = (Direction)180;
int degree = (int)home;
```

```
// funktionale Notation
int x = 3, y = 2;
double r = double(x) / double(y);
```

```
// funktionale Notation
Direction home = Direction(180);
int degree = int(home);
```

```
// implizit
int n = 1;
double x = n;
```

```
// C++
double x = 1.5;
int n = static_cast<int>(x);
```

```
// bitweise Konvertierung (extrem selten notwendig)
char header[4] = {'R', 'I', 'F', 'F'};
unsigned int& id = reinterpret_cast<unsigned int&>(header);
```

```
// zu vermeiden (zeugt von unsauberem Code)
const double pi = 3.14159;
double* p = const_cast<double*>(&pi);
```

Aufgaben

Aufgabe 15A

- ein double soll auf verschiedene Weise in einen int umgewandelt werden

Aufgabe 15B

- Wie könnte man Fließkommazahlen per Typumwandlung runden?
- testen mit 1.2, 2.5, 3.7 und -0.6
- vergleichen mit den Funktionen `round`, `floor`, `ceil` und `trunc` aus `<cmath>`

Referenzen

```
int a = 1;
int* p = &a;
*p = 2;
cout << a << endl;

int b = 3;
p = &b;
*p = 4;
cout << a << b << *p;
// 2 4 4
```

```
int a = 1;
int& r = a;
r = 2;
cout << a << endl;

int b = 3;
r = b;
r = 4;
cout << a << b << r;
// 4 3 4
```

- ähnlich wie Pointer verweisen Referenzen auf andere Variablen
- nur einmalige Zuweisung möglich (und nötig!)
- keine Nullreferenzen (anders als Nullpointer)

Referenzen

```
void swap(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}

int main()
{
  int x = 4;
  int y = 17;
  swap(x, y);
  return 0;
}
```

```
#include <utility>
std::swap(x, y);
```

C++ 11

```
void swap(int* a, int* b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main()
{
  int x = 4;
  int y = 17;
  swap(&x, &y);
  swap(&x, nullptr);
  return 0;
}
```

Referenzen

```
void print(const int number)
{
  cout << number << endl;
}

void print(const string& text)
{
  cout << text << ": ";
}

int main()
{
  print("die Antwort lautet");
  print(42);
  return 0;
}
```

Aufgabe

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

using std::string;
using std::cout;
using std::endl;

void print(const string& text)
{
  cout << text << endl;
}

int main()
{
  string text = "warum?";
  print(text);
  return 0;
}
```

Aufgabe 16

- Was passiert, wenn man das „const“ entfernt?
- Was passiert, wenn man die String-Variable entfernt und den Text direkt an die Funktion übergibt?
- Was passiert, wenn man die Funktion den Text nicht als Referenz, sondern als Wert entgegennimmt?

Referenzen

```
string& getErrorMessage()
{
  string result = "Das ist ein Fehler.";
  return result;
}

double* calcPi()
{
  double result = atan(1.0) * 4.0;
  return &result;
}
```

Referenzen auf lokale Variablen sind gefährlich!

Pointer auf lokale Variablen sind nicht ungefährlicher!

- Referenzen und Pointer auf lokale Variablen bei Rückgabewerten vermeiden
- unter Umständen eignen sich „unique pointer“ oder „shared pointer“