Teil 4

Zeiger und Arrays

- Typinferenz
- Konstanten
- Zeiger
- Heap
- Nullpointer
- Smartpointer
- Arrays
- C-Strings

Typinferenz

C++ 11

```
auto count = 12;
auto area = 17.2;
std::string text = "hallo";
char* text = "hallo";
auto correct = true;
auto letter = 'X';

auto next = count + 1;
auto result = foo(1, 2, 3);
```

- Typinferenz findet zur Compile-Zeit statt
- statische, keine dynamische Typisierung !!!

Compiler-Flags

C++ 11

- Kommandozeile GCC und Clang

```
g++ -std=c++11 -c main.cpp
```

- Projektdatei Qt Creator (.pro)

```
TEMPLATE = app
CONFIG += c++11
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp
```

- für C++14 entweder c++1y oder c++14
- für MSVC gibt es /std:c++14 oder /std:c++latest

Konstanten

```
const int answer = 42;
++answer;
```

Fehler zur Compile-Zeit

```
int const answer = 42;
```

- Schutz vor versehentlicher Veränderung
- Optimierungspotential für den Compiler

```
bool isDividable(const int a, const int b)
{
  const int r = a % b;
  return r == 0;
}
```

Aufgaben

Aufgabe 10A

- die Funktionen für Potenz, Fakultät und Binomialkoeffizient sollen mit möglichst vielen Konstanten versehen werden

Aufgabe 10B

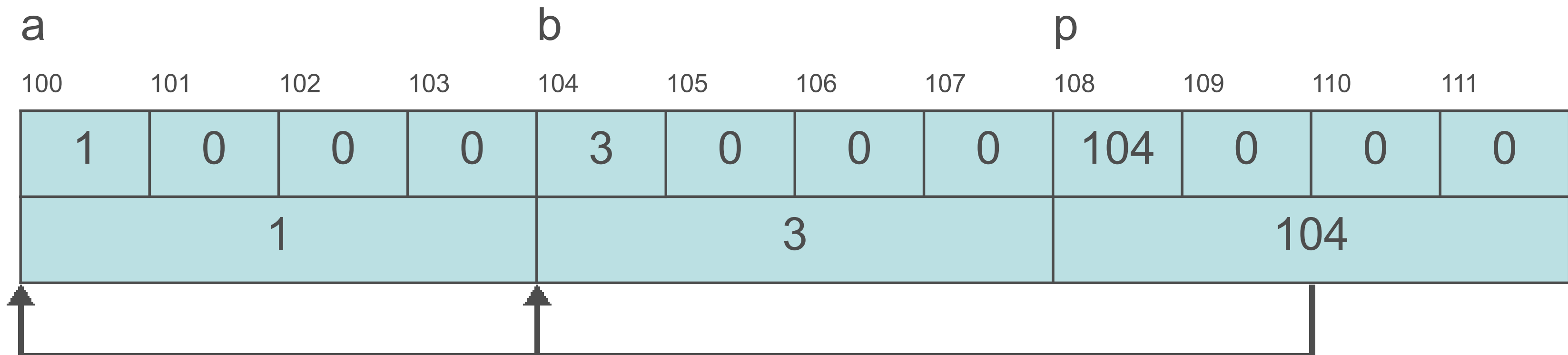
- die Funktionen für Teiler und Primzahlen sollen mit möglichst vielen Konstanten versehen werden

```
bool isDividable(const int a, const int b)
{
  const int r = a % b;
  return r == 0;
}
```

Zeiger

- Zeiger beinhalten Arbeitsspeicheradressen

```
int a = 1;
int b = 2;
int* p = &a;
p = &b;
*p = 3;
```



Zeiger

```
int number = 73;
int* pointer = &number;
*pointer = 74;
cout << number << endl;
```

- Typ + `*` bezeichnet einen Zeigertyp, der auf eine (möglichst sinnvolle) Adresse im Arbeitsspeicher verweist
- Zeiger in C++ sind typisiert (Ausnahme: `void*`)
- ein vorangestelltes `&` referenziert eine Variable (Zugriff auf Adresse der Variablen)
- ein vorangestelltes `*` dereferenziert eine Variable (Zugriff auf Wert der Zeigervariablen)

konstante Zeiger

```
int v = 1;
const int c = 2;

int* p1 = &v;
p1 = &c;
*p1 = 3;

const int* p2 = &v;
p2 = &c;
*p2 = 4;

int* const p3 = &v;
p3 = &c;
*p3 = 5;

const int* const p4 = &c;
```

falsch, c ist „const“

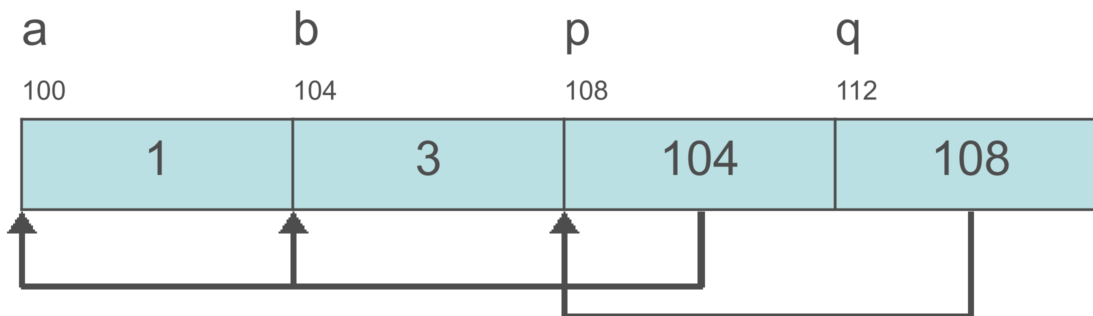
falsch, p2 ist „const“

falsch, p3 zeigt fest auf v

Zeiger auf Zeiger

- Zeiger können beliebig verschachtelt werden

```
int a = 1;
int b = 2;
int* p = &a;
int** q = &p;
*q = &b
**q = 3;
```



Der Heap

```
int* pointer = new int(73);
*pointer = 74;
cout << *pointer << endl;
delete pointer;
```

- `new` erzeugt ein Objekt des angegebenen Typs auf dem *Heap* (dynamische Speichervergabe) - dort verbleibt es bis es manuell aufgeräumt wird
- die Zeigervariable, die auf das Objekt verweist, liegt ganz normal auf dem *Stack*
- `delete` löscht ein zuvor erzeugtes Objekt (immer genau einmal aufrufen!)
- `new` und `delete` kommen immer paarweise vor

Aufgaben

Aufgabe 11A

- eine Funktion „changeNumber“ soll den Wert eines übergebenen Int-Pointers ändern
- die Main-Funktion soll die Funktion aufrufen, um den Wert einer Variablen zu ändern
- Lässt sich der Wert des Ints innerhalb der Funktion „changeNumber“ auch ohne Pointer ändern?

Aufgabe 11B

- eine erste Funktion „createNumber“ soll einen neuen int auf dem Heap anlegen, mit einem Anfangswert belegen und einen Pointer darauf zurückliefern.
- eine zweite Funktion „changeNumber“ soll den Wert eines übergebenen Int-Pointers ändern
- die Main-Funktion soll die beiden Funktionen aufrufen, den Wert des Pointers ausgeben, und am Ende löschen

Nullpointer

```cpp
int* pointer = nullptr;
if (pointer)
{
  *pointer = 74;
}
delete pointer;
```

- Nullpointer zeigen quasi nirgendwohin
- beim Dereferenzieren eines Nullpointers kracht's
- seit C++11 gibt es `nullptr`
- zuvor nutzte man `0` oder besser `(void*)0`
- Nullpointerkonstanten können in jeden Pointertyp konvertiert werden

Smartpointer

C++ 11

```
#include <memory>

{
  std::unique_ptr<int> pointer = new int(73);
  *pointer = 74;
  cout << *pointer << endl;
  // kein delete!
}
```

Template-Parameter

- der `unique_ptr` nimmt Besitz über den Zeiger
- das Objekt wird automatisch gelöscht, sobald die (eine) Zeigervariable *out of scope* geht
- es wird (mehr oder weniger) sichergestellt, dass keine anderen Pointer auf dieselbe Variable zeigen
- veraltet: `auto_ptr`

Smartpointer

C++ 11

```
#include <memory>

std::shared_ptr<int> a = new int(73);
std::shared_ptr<int> b = a;
*a = 74;
cout << *b << endl;
// kein delete!
```

- das Objekt wird automatisch gelöscht, sobald die letzte Zeigervariable, die es referenziert, *out of scope* geht
- funktioniert mit Referenz-Counter
- funktioniert nicht bei zirkulären Referenzen
- beugt *Memory Leaks* vor, kann diese aber nicht gänzlich ausschließen

Smartpointer

C++ 11

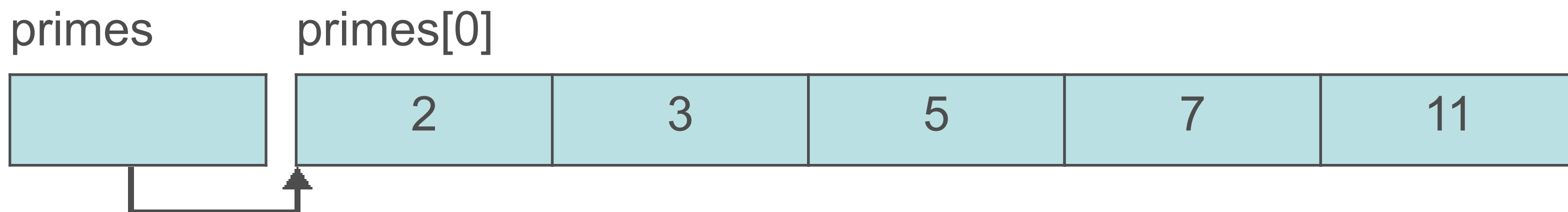
```
#include <memory>

auto pointer = std::make_shared<int>(73);
*pointer = 74;
cout << *pointer << endl;
// kein delete!
```

- `make_shared` empfohlen, da effizienter (1 statt 2 Speicherallokationen)
- und weniger fehleranfällig (bei Exceptions)
- analog `std::make_unique`

Arrays

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



- das erste Element hat stets den Index 0
- Array-Variablen sind im Prinzip Pointer auf das erste Element

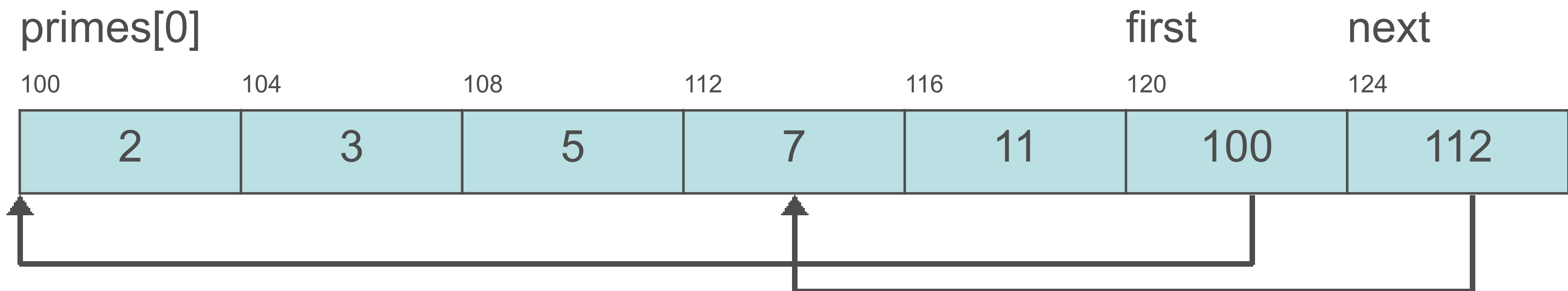
```
int* first = primes;
cout << *first << endl;
cout << first[1] << endl;
cout << *(first + 1) << endl;
```

```
int nulls[5] = {};
```

Zeigerarithmetik

```
int primes[] = {2, 3, 5, 7, 11};
int* first = primes oder alternativ &primes[0];
int* next = first + 1;
next += 2;
cout << *next << endl;
```

- brücksichtigt Größe des Datentyps
- schnell, aber nicht immer deutlich schneller
- gefährlich (Speicherverletzungen / -zugriffsfehler)



Arrays

```
int magicSquare[3][3] = {{4, 9, 2}, {3, 5, 7}, {8, 1, 6}};
cout << primes[1][1] << endl;
```

- mehrdimensionale Arrays sind im Prinzip „Arrays aus Arrays“
- dynamische Arrays mit `new`

```
int* numbers = new int[n];
for (int i = 0; i < n; ++i) numbers[i] = 2 * i;
delete [] numbers;
```

Klammern nicht vergessen !!!

C-Strings

```
char text[] = "hallo";
```

```
char text[] = {'h', 'a', 'l', 'l', 'o', '\0'};
```

'h'	'a'	'l'	'l'	'o'	#0

- sind Arrays
- kennen ihre Länge nicht direkt
- sind null-terminiert

```
char* text = "hallo";
char* texts[] = {"eins" , "zwei", "drei"};
```

Arrays

```
int main(int argc, char* argv[])
{
  return 0;
}
```

Aufgabe 12A

- die übergebenen Parameter sollen ausgegeben werden (einer pro Zeile)
- Was enthält `argv[0]`?

Aufgabe 12B

- zwei übergebene Zahlen sollen addiert werden (`atoi(s)` aus `<cstdlib>` holt einen int aus einem C-String, `std::stoi(s)` aus einem std::string)

- optional: Fehlerbehandlung

C-Strings

```cpp
#include <cstring>

char text[100];
const char* word = "C-Strings";

strcpy(text, word);
strcat(text, " sind ");
strcat(text, word);

int length = strlen(text);
int pos = strstr("in");
```

- unhandlich
- teils langsam
- nicht geschützt vor **Buffer-Overflows** !!!
- können kein Nullzeichen enthalten

```cpp
#include <string>

using std::string;

string text;
string word = "STD-Strings";

text = word + " sind " + word;

int length = text.length();
int pos = text.find("in");
```

- können Nullzeichen enthalten
- Literale sind C-Strings (seit C++11 gibt es auch STD-String-Literale `"abc"s`)