Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

15. November 2019

Ergänzende Themen

Inhalt

- Zeichen- und Literal-Typen
- Reguläre Ausdrücke
- Wahrscheinlichkeitsverteilungen
- Dateisystem
- Parallelität

Neue Zeichentypen

bisher (meistens ASCII)

```
const char* s    =  "abcd";     // 8-Bit String
const wchar_t* s = L"abcd";     // 16-Bit String
```

neu (Unicode)

```
const char* s     = u8"abcd";   // UTF8 String-Repräsentation
const char16_t* s =  u"abcd";   // UTF16 String-Repräsentation
const char32_t* s =  U"abcd";   // UTF32 String-Repräsentation
```

Unicode-Codepoints

```
16 Bit Unicode-Codepoints:  \u1234      (4-stelliger Hex-Code)
32 Bit Unicode-Codepoints:  \u123456    (6-stelliger Hex-Code)
```

Rohstring-Literale

- Mit normalen Strings ist die Benutzung von bestimmten Zeichen schwierig
  - Beispiel: Anführungszeichen, Backslashes und Sonderzeichen müssen speziell markiert werden: `"abc \"def\" \\ghi\\\n"`
- Mit Rohstringliteralen kann das Problem umgangen werden
- Syntax: `R"<eigener Marker>(text)<eigener Marker>"`
- Die Kombination aus `"(` resp. `)"` und dem eigenen Marker darf im Text weiterhin nicht vorkommen

```
1  const char* s = R"(My dog's name is "chappy".)";
```

```
1  const char* s = R"--(<HTML>
2  <HEAD>
3  <META HTTP-EQUIV="Content-Type">
4  )--";
```

Benutzerdefinierte Literale

- Selbst festgelegtes Suffix, welches einem Literal (Zeichenkette oder Ziffernfolge) nachgestellt werden kann
- Suffix muss mit einem _ beginnen
- Implementation mittels Literal-Operator: operator""

```
1  Complex operator"" _i(double imag) {
2    return Complex(0, imag);
3  }
4
5  Complex c = 2.5_i; // Komplexe Zahl 0 + 2.5 i
```

- Interpretation des Literals vor dem Suffix möglich als:
  - Dezimal-Literal: `uint8_t`, `int`, `bool`, etc.
  - Floating-Point-Literal: `float`, `double`
  - Character-Literal: `char`
  - String-Literal: `const char*`, `(const char*, size_t)`

Reguläre Ausdrücke

- regular expressions in Header `regex` implementiert
- Syntax ist by default: ECMAScript
- Kombination mit Rohstring-Literalen sinnvoll
- Wiederverwenden der Matches mit std::regex_iterator

```
#include <regex>
static const std::regex natel(R"(07[689]\d{7,7})");
```

```
bool IsNatel(const std::string& s) {
  return std::regex_match(s, natel);
}

bool ContainsNatel(const std::string& s) {
  return std::regex_search(s, natel);
}
```

Wahrscheinlichkeitsverteilungen

Zufallszahlen-Generator mit 20 unterschiedlichen Wahrscheinlichkeitsverteilungen

```
1  #include <random>
2
3  // würfelt Zahl zwischen 0..5
4  uniform_int_distribution<int> distribution(0, 5);
5
6  default_random_engine e;
7
8  std::array<size_t, 6> counts;
9  for (int i = 0; i < 100000; ++i) {
10   ++counts[distribution(e)];
11 }
12
13 for (auto c: counts) {
14   std::cout << c << std::endl;
15 }
```

Dateisystem

Handling von Pfad-Namen und Manipulation von Dateien etc.

```cpp
#include <iostream>
#include <filesystem>
#include <fstream>
namespace fs = std::filesystem;

fs::create_directories("sandbox/dir");
std::ofstream("sandbox/file1.txt").put('a');
fs::copy("sandbox/file1.txt", "sandbox/dir/file2.txt");
fs::copy("sandbox/dir", "sandbox/dir2");  // nicht-rekursives Kopieren
  // sandbox/dir
  // sandbox/dir/file2.txt
  // sandbox/dir2
  // sandbox/file1.txt
fs::copy("sandbox", "sandbox/copy", fs::copy_options::recursive);
// kopiert rekursive den kompletten Ordner sandbox/
fs::remove_all("sandbox");
```

Threads

- `std::thread` spawnt einen separaten Thread
- startet das ausführbare Objekt mit den entsprechenden Parametern:
  - Funktor
  - Lambda
  - etc.
- das ausführbare Objekt und die Parameter werden standardmässig kopiert

```
1  auto lambda = [](size_t count) { ... };
2  std::thread t(lambda, 24);  // ähnlich wie std::bind
3  ...
4  t.join();
```

Threads und Futures

- Thread
  - low-level
  - Austausch von Daten muss selber synchronisiert werden
  - nicht abgefangene Exceptions in der Thread-Funktion führen zum Abbruch des gesamten Programms
  - `thread_local` Speicherklasse: statische/globale Variablen werden pro Thread angelegt
- Future
  - Asynchrone Verarbeitung: parallel oder erst beim Aufruf von `get()`
  - Exceptions tauchen im Vater-Thread auf, wenn das Ergebnis mit `get()` abgeholt wird
  - wird der Scope des verantwortlichen Futures verlassen, so sorgt der Destruktor dafür, dass die Berechnung problemlos zu Ende geführt wird

Asynchrone Ausführung

- `std::async`
  - z.B. parallel zu anderen Arbeiten
  - oder nicht beim Starten von `std::async()`, sondern erst beim Aufruf von `get()`
- Rückgabewert von `async()`
  - `std::future<RT>`, wobei `RT` der Rückgabetyp der asynchron ausgeführten Funktion ist
- Launch Policy (Ausführungsrichtlinie)
  - `std::async(std::launch::async, berechnung)`: garantiert parallele Ausführung
  - `std::async(std::launch::deferred, berechnung)`: Ausführung bei `get()`
- ein Future kann auch ohne `std::async` erzeugt werden, dazu muss zuerst ein `std::promise` (eine Art Übertragungskanal) erstellt werden

Synchronisierter Zugriff

- `std::atomic`: alle Zugriffe sind atomar (werden nicht unterbrochen)
- `std::atomic_flag`: atomarer bool, jedoch lock-free
- `std::once_flag`: verwendet in call_once, stellt sicher dass nur einer der parallelen Threads die Funktion ausführen wird
- `std::mutex`: ermöglicht wechselseitigen Ausschluss
- `std::recursive_mutex`: ermöglicht den gleichen Thread mehrfach in den kritischen Abschnitt einzutreten
- `std::lock_guard`: schützt einen kritischen Abschnitt, sehr einfache Anwendung, kennt nur den Zustand locked
- `std::unique_lock`: braucht sein eigenes Mutex, kennt locked und unlocked
- `std::condition_variable`: blockiert den Thread bis er von einem anderen Thread ein Signal zur Fortsetzung erhält