Kapitel 28

28.1 Die Operatoren `*` und `++` tun *nichts* außer `*this` zurückzugeben, siehe Abschnitt 28.4.

Kapitel 29

29.1 *cppbuch/loesungen/k29/1.cpp*

`count()` würde nicht funktionieren, weil es ein `pair`-Objekt als Parameter verlangt, es aber in der Aufgabe nur um den Rang, also nur einen Teil der Paar-Kombination geht. Mit `count_if()`, das ein Prädikat verlangt (vgl. Seite 737), ist das Problem zu lösen, weil das Prädikat beliebig gestaltet werden kann. Das Prädikat ist ein Funktionsobjekt und vergleicht nur, ob der Rang der gewünschte ist – der Name wird ignoriert.

29.2 *cppbuch/loesungen/k29/2.cpp*

`equal_range()` arbeitet nur auf sortierten Containern und braucht daher die Information, welches von zwei Elementen das größere ist. In diesem speziellen Fall wird dabei nur der Rang verglichen. Aus denselben Gründen wie in der vorhergehenden Lösung benötigt `equal_range()` ein Funktionsobjekt, das den Vergleich erledigt. Im Lösungsvorschlag zeigt sich wieder, wie praktisch `auto` ist. Ohne dieses Schlüsselwort müsste

```
pair<multimap<int, string, greater<int> >::iterator,
     multimap<int, string, greater<int> >::iterator>
bereich = equal_range(promis.begin(), promis.end(),
                      gesuchtesPaar, Rangvergleich());
```

geschrieben werden.

Kapitel 32

32.1 *cppbuch/loesungen/k32/*

A.7 Änderungen in der 6. Auflage

Um den Seitenumfang in etwa beizubehalten, wurden manche Abschnitte gekürzt. Das Kapitel über die wenig benutzten `valarrays` ist ganz weggefallen, wird aber noch im Internet angeboten (*http://www.cppbuch.de/valarrays.pdf*). Sämtliche Beispiele wurden an die Möglichkeiten des neuen Standards angepasst, sofern sinnvoll. Viele Kapitel und Abschnitte wurden teilweise oder in Gänze überarbeitet. Es gibt neue Abschnitte, so zum Programmierstil und natürlich zu den neuen Features von C++20. Die unten aufgeführten Änderungen in C++20 gegenüber C++17 werden im Buch berücksichtigt. Die Änderungen sind teilweise gering, teilweise umfangreich. Aus Platzgründen wurden Änderungen,

die im Wesentlichen nur für Experten interessant sind, nicht berücksichtigt. Auch wird in einigen Fällen die grundsätzliche Eigenschaft einer Änderung beschrieben, ohne auf Details einzugehen.

- `atomic_ref` Template für atomare Operationen mit dem referenzierten Objekt.
- *Concepts* prüfen die Compilationsvoraussetzungen für Templates mit dem Ergebnis verständlicherer Fehlermeldungen und verkürzten Übersetzungszeiten.
- `char8_t` ist ein Typ für UTF8-Zeichen (ohne Multi-Byte-Sequenzen). Der dazugehörende Stringtyp ist `std::u8string`.
- `consteval` erzwingt die Auswertung einer Funktion zur Compilationszeit.
- `constexpr` ist an sehr viel mehr Stellen als bisher einsetzbar mit dem Vorteil der Laufzeitersparnis. Unter anderem sind jetzt `std::vector`, `std::string` und viele Algorithmen `constexpr`.
- `constinit` vermeidet das Problem der Reihenfolge statischer Initialisierungen.
- Das `contiguous_iterator`-Konzept garantiert, dass die angesprochenen Elemente im Speicher direkt nacheinander liegen.
- *Coroutinen* sind Funktionen, die angehalten und wieder fortgesetzt werden können.
- `for`-Schleifen, die auf einem Bereich basieren, können mit einer Variable erweitert werden, wie die Zählvariable `i` im Beispiel `for (int i = 0; const auto& el : container) { ... }`
- *Elementnamen* können bei der Initialisierung direkt benannt werden, zum Beispiel `Punkt p { .x = 10, .y = 20 }`.
- `iterator_traits<I>` können jetzt kürzer geschrieben werden.
- `jthread` vereinfacht die Programmierung von Threads.
- *Lambda*-Funktionen können Template-Parameter übergeben werden. `[=]` soll nicht mehr verwendet werden (Empfehlung: `[=, this]`).
- *Module* sind langfristig ein Ersatz für die herkömmliche Einbindung anderer Programmteile mit Libraries und `#include`.
- *Negative Ganzzahlen* werden als Zweierkomplement realisiert.
- `<numbers>` ist ein Header für mathematische Konstanten wie π, e und viele andere.
- Bereiche (`std::ranges`) (hat Auswirkungen auf die meisten Algorithmen).
- `source_location` soll die Makros `__FILE__`, `__LINE__` usw. ersetzen.
- `std::span` ist eine Sicht auf einen Container mit aufeinanderfolgenden Objekten, ähnlich wie `string_view` die Sicht auf einen `string` ist.
- `ssize()` und `ranges::ssize()` sind vorzeichenbehaftete Funktionen, die in vielen Fällen als Ersatz für `size()` dienen, um den Typkonflikt `signed`/`unsigned` zu vermeiden.
- `std::string` hat jetzt auch `starts_with()`- und `ends_with()`-Methoden.
- Die *Textformatierung* mit `std::format` ist erheblich einfacher als die bisher angebotene mit `width`, Manipulatoren u.a.
- `using enum` erspart Schreibarbeit bei der Abfrage mit `switch`/`case`.
- `<=>` ist ein Operator, der automatisch andere Vergleichsoperatoren definieren kann.
- `lexicographical_compare_three_way` ist eine dazu passende Vergleichsfunktion.