C++: Schnell und klein

Was kostet ein Sprach-Feature?

Andreas Fertig

Mit dem Standard C++ 11 und seinen Nachfolgern können C++-Entwickler von spannenden neuen Eigenschaften der Sprache profitieren. Features sind stets mit Kosten verbunden, entweder Geschwindigkeit oder Code-Größe. Um die Sprache C++ mit dem Motto "you pay only for what you use" effektiv nutzen zu können, ist es für Entwickler wichtig zu wissen, was die jeweiligen Kosten eines Features sind.

Die Programmiersprache C++ ist dafür bekannt sowohl objektorientiert als auch effizient zu sein. Es gilt das Motto „you pay only for what you use“. Wir können Sprach-Features in unserem Code verwenden und zahlen damit für ihren Einsatz. Gleichzeitig profitieren wir von möglichen Vorteilen. Verwenden wir bestimmte Features nicht, entfallen die Kosten dafür. Wir können somit in Maßen die Kosten steuern. Kosten bedeutet erhöhter Speicherverbrauch (RAM oder ROM) oder Laufzeit. Eine Kombination aus beidem ist ebenfalls möglich.

Temporäre-Objekte sind ein Beispiel. Die Sprache hilft uns an dieser Stelle Konvertierungen von einem Typ in einen anderen vorzunehmen. Eine Konvertierung lässt sich auch klarer Ausdrücken, dann entfallen die Kosten für die Erstellung und Zerstörung des temporären Objekts.

Wir hatten viele Jahre, von 1999 bis 2011, Zeit uns Wissen über die Kosten von Sprach-Features anzueignen. Mit dem neuen Standard C++11 und seinen Nachfolgern sind der Sprache eine Vielzahl von Neuerung widerfahren. Mit diesen Neuerungen von Kosten und Vorteile verbunden die für den Einsatz eines Features wissenswert sind.

Schauen wir auf `auto`. Mit diesem Schlüsselwort überlassen wir dem Compiler die Bestimmung des Datentyps einer Variablen. Hier zapfen wir schlicht das schon immer existierende interne Wissen des Compilers an. Wir bitten ihn darum den Typen des Ausdrucks auf der rechten Seite auf die Linke zu übertragen. Da C++ keine dynamische Sprache ist, steht das Wissen um den Typ zur Compile-Zeit fest. Es ergibt sich kein Laufzeitnachteil. Mit `auto` können wir längliche Template-Ausdrücke reduzieren und Schreibarbeit dem Compiler überlassen. In einigen Fällen hilft uns `auto` da es garantiert, dass links und rechts der gleiche Typ steht. So lassen sich Flüchtigkeitsfehler wie `const` rechts aber nicht links, was zu einem temporären Objekt führen kann, vermeiden.

Neben `auto` wurde C++ um sog. range-based for-Schleifen, in anderen Sprachen auch als foreach-Schleifen bezeichnet, erweitert. Sie dienen uns dazu klareren und reduzierten Code zu schreiben. Eine Schleife um über einen Vektor zu iterieren lässt sich vor C++11 so formulieren:

```
std::vector<int> numbers{1, 2, 3, 5};

for(auto it = numbers.begin(); it != numbers.end(); ++it)
```

```
{
  printf("%d\n", *it);
}
```

Hier ist es notwendig Details des Typen wie `begin` und `end` zu kennen. Davon abgesehen ist der Code im Kopf der for-Schleife wenig überraschend oder fordernd für einen selbst wenig erfahrenen EntwicklerIn. Der Code ist eher lästig zu schreiben und zu lesen. Mit C++11-Mitteln überlassen wir auch hier wieder das Ausfüllen dem Compiler und reduzieren unsere Schreibarbeit auf das Wesentliche. In Verbindung mit `auto` können wir dieses wesentliche noch weitere reduzieren. Die gleiche Schleife sieht in C++11 dann so aus:

```
std::vector<int> numbers{1, 2, 3, 5};

for(auto it : numbers)
{
  printf("%d\n", it);
}
```

Wesentlich kompakter zu schreiben und zu lesen. Was jedoch geschieht im Hintergrund um den gleichen Effekt wie zuvor zu erzielen? Und wie effizient ist das was der Compiler für uns an dieser Stelle tut?

Es handelt sich um C++ und das Standardisierungskomitee ist stets bemüht auch die neusten Feature effizient zu gestalten. So auch in diesem Fall. Im Standard [1] ist unter [stmt.ranged] festgelegt wie eine range-based for-Schleife aussieht:

```
auto && __range = range-init;

for ( auto __begin = begin-expr,
           __end   = end-expr;
      __begin != __end;
      ++__begin ) {
  for-range-declaration = *__begin;
  statement
}
```

An dieser Stelle nutzt der Standard intern `auto` zur Ermittlung des Range-Typs. Ansonsten baut eine range-based for-Schleife darauf, dass es entweder drei Funktionen in einer Klassen gibt die den Beginn und das Ende liefern sowie einen Operator zum Vergleich auf Ungleichheit. Der Rest ist eine altbekannte for-Schleife.

Eine range-based for-Schleife bietet noch ein wenig mehr als die simple Schreibvereinfachung. Zum einen verwendet der Standard den Pre-Inkrement anstelle des ebenfalls möglichen Post-Inkrements. Dies vermeidet das temporäre Objekt das bei einem Post-Inkrement vorkommt. Weiter wird mit der Variable `__end` ein harter Cache der Schleifenendbedigung aufgebaut. Damit kann sich ein Geschwindigkeitsvorteil ergeben, da nicht in jedem Schleifendurchlauf erneut `end()` aufgerufen wird. Selbstverständlich lassen sich damit keine etwaigen Tricks innerhalb von `end()` mehr umsetzen.

Insgesamt lässt sich sagen, dass auch mit den neuen Standards C++ seiner Linie treu bleibt. Nur Features die verwendet werden schlagen mit Kosten zu buche. Der

Standard achtet darauf mit jedem Feature auch einen wirklichen Gewinn zu bringen. Dies gilt auch für hier nicht betrachtete Features wie Lambdas oder structured bindings. Ungeachtet dessen lohnt es sich in Sachen Kosten auf dem neuesten Stand zu bleiben.

Literatur- und Quellenverzeichnis

[1] Working Draft, Standard for Programming Language C++ (N4687)

Autor

Andreas Fertig studierte Informatik in Karlsruhe. Bereits seit seinem Studium befasst er sich mit eingebetteten Systemen und den damit einher gehenden Anforderungen und Besonderheiten. Seit 2010 ist er für die Philips Medizin Systeme als Softwareentwickler mit dem Schwerpunkt eingebettete Systeme tätig. Er verfügt über fundierte praktische und theoretische Kenntnisse von C++ auf verschiedenen Betriebssystemen. Freiberuflich arbeitet er als Dozent und Trainer. Zudem entwickelt er verschiedene Mac OS X Anwendungen.

Kontakt

Internet: https://www.andreasfertig.info
Email: contact@andreasfertig.info