C++: Typsicher

Fehler zur Compile-Zeit finden und Tippfehler reduzieren

Andreas Fertig

Templates existieren in C++ bereits seit einiger Zeit. Mit dem C++11 Standard-Update wurden sie noch besser. Nun gibt es variadische Template-Argumente. Einige Leute argumentieren, dass dies das wichtigste neue Feature in C++11 ist. Templates sind eine gute Möglichkeit, den Compiler Code für Sie erstellen zu lassen. Da Templates zur Kompilierzeit ausgewertet werden, sind sie auch perfekt für eine frühestmögliche Fehlererkennung. Sie können robusteren Code mit ihnen schreiben. Mit Templates können Sie Berechnungen bereits zur Kompilierzeit durchführen. Zusammen mit constexpr sind sie ein sehr leistungsfähiges Werkzeug, das jeder in seiner Werkzeugkiste haben sollte.

C++ verfügt über ein sehr starkes Typsystem. Zugegeben, dass es oft unterwandert oder ignoriert wird. In vielen Bereichen, in denen C++ eingesetzt wird, spielt jedoch gerade diese Typsicherheit eine entscheidende Rolle. Es geht darum, bereits zur Compile-Zeit zu erkennen, dass z. B. eine Zuweisung invalide ist oder ein Parameter nicht zu einem Funktionsaufruf passt.

„Type safety means that *the compiler will validate types while compiling, and throw an error if you try to assign the wrong type to a variable*.“ [1]

Ein Beispiel für eine typunsichere Funktion ist dieses Beispiel:

```
void Foo(bool first, bool addNewLine);

void UseFoo(bool first, bool addNewLine)
{
  Foo(addNewLine, first);
}

void Main()
{
  UseFoo(true, false);
}
```

Zunächst ist es schwer zu sagen, was die beiden Parameter bewirken, wenn wir lediglich den Aufruf `UseFoo(true, false)` betrachten. Inspizieren wir die Implementierung näher, können wir erkennen, dass innerhalb von `UseFoo` die beiden Parameter `first` und `addNewLine` vertauscht wurden. Ob das korrekt ist oder nicht, ist auch an dieser Stelle schwer zu sagen. Erst die Parameternamen in der Implementierung von `Foo` deuten an, dass hier ein Fehler vorliegt und die Parameter tatsächlich vertauscht wurden. Für den Compiler gibt es hier absolut keinen Grund, eine Warnung zu generieren. Beide Parameter sind vom Typ bool, also alles bestens. Die Auswirkung auf unser Programm kann jedoch katastrophal sein. Das schlimmste ist, wir sehen die Auswirkung erst durch das Ausführen des Programms. Damit bestimmt die Güte unseres Testings, ob der Fehler noch in der Produktion oder erst beim Kunden gefunden wird.

Mit lediglich einem kleinen Trick können wir für mehr Sicherheit sorgen und diese Art von Fehler bereits zur Compile-Zeit finden. Also völlig unabhängig von unserem Testing:

```
#define STRONG_BOOL(typeName)                           \
  enum class typeName : bool                            \
  {                                                     \
    No  = false,                                        \
    Yes = true                                          \
  }

STRONG_BOOL(First);
STRONG_BOOL(AddNewLine);

void Foo(First first, AddNewLine addNewLine);

void UseFoo(First first, AddNewLine addNewLine)
{
  // Foo(addNewLine, first);
  Foo(first, addNewLine);
}

void Main()
{
  UseFoo(First::Yes, AddNewLine::No);
}
```

Wir erstellen uns einen Klassen-Enum, der zwei Werte enthält: Yes und No. Das Makro dient dazu, einfach einen solchen Klassen-Enum erstellen zu können. Als Beispiel sehen wir `First` und `AddNewLine`. Diese beiden neuen Typen nutzen wir anschließend in der Funktionssignatur von `Foo` und `UseFoo`. Damit ist ein Vertauschen der Parameter nicht mehr möglich. Der Compiler wird mit einem harten Fehler den Compile-Vorgang abbrechen. Als weiteres Plus ist der Aufruf von UseFoo viel klarer. Jetzt ist direkt zu erkennen, wofür die beiden Parameter stehen.

Templates können sehr hilfreich sein, wenn es um Typsicherheit geht. Schauen wir uns dazu folgendes Beispiel an:

```
int16_t max(int16_t a, int16_t b)
{
    return (a > b) ? a : b;
}

int main()
{
    int16_t  a = 1;
    uint16_t b = 65530;

    printf("max: %d\n", max(a, b));
}
```

Hier sehen wir eine Implementierung der max-Funktion. Diese arbeitet mit 16-Bit vorzeichenbehafteten Integern. Im `printf`-Aufruf übergeben wir die beiden

Parameter `a` und `b`. Das Programm ist klein und übersichtlich. Wir erkennen unschwer, dass wir an dieser Stelle einen vorzeichenlosen Wert mit dem Parameter `b` übergeben. Macht nichts, es compiliert und linked. Was aber ist das Ergebnis? Wir erwarten 65530, korrekt? Die Ausgabe von `printf` wird uns mit 1 vermutlich erstaunen. Hier kommen die Integer-Konvertierungsregeln zu tragen und aus dem unsigned wird ein sehr kleiner signed.

Templates können hier von entscheidender Hilfe sein. Schauen wir uns dazu das leicht modifizierte Programm an:

```
template<typename T>
T max(T_t a, T_t b)
{
    return (a > b) ? a : b;
}

int main()
{
    int16_t  a = 1;
    uint16_t b = 65530;

    printf("max: %d\n", max(a, b));
}
```

Die `max`-Funktion wurde lediglich durch den Einsatz eines Templates generalisiert. Der Effekt, den wir durch das Template erhalten, ist, dass dieses Programm mit einem Compile-Fehler endet. Templates sind typsicher und die Signatur des Templates erfordert zwei Parameter vom gleichen Typ. Hier kommen keine Konvertierungsregeln zum Einsatz. Denn beide Typen müssen direkt passen. Zu beachten ist, dass Fließkommazahlen hier gegf. gesondert zu betrachten sind.

Templates sind ein einfaches Mittel, um die Typsicherheit von C++ weiter zu erhöhen und Fehler bereits zur Compile-Zeit zu erkennen.

Literatur- und Quellenverzeichnis

[1] Razin, “What is type-safe?”. https://stackoverflow.com/questions/260626/what-is-type-safe

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 Kenntnisse von C++.

Freiberuflich arbeitet er als Dozent und Trainer. Zudem entwickelt er Mac OS X-Anwendungen und ist der Autor von cppinsights.io.

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